Collège Doctoral Languedoc-Roussillon

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **UR GREEN - Cirad**

Spécialité: **Informatique**

Présentée par **Paulo Pimenta**

Application of Model-driven engineering to multi-agent systems: a language to model behaviors of reactive agents

Soutenue le 5 Janvier 2017 devant le jury composé de:

M. Jacques Ferber	Professseur	Université de Montpellier	Pres. du Jury
M. Jean-Pierre Müller	HDR	CIRAD	Dir. de thèse
M. David Hill	Professseur	Université Blaise-Pascal	Rapporteur
M. Jean-Michel Bruel	Professseur	Université de Toulouse	Rapporteur
M. Mamadou Kaba Traore	Mâitre de conference/HDR	Université Blaise Pascal	Examinateur
M. Fabien Michel	Mâitre de conference/HDR	Université de Montpellier	Examinateur
M. Jaime Simao Sichman	Mâitre de conference	Universidade de São Paulo	Invité
M. Pierre Bommel	Docteur	CIRAD	Invité

Université Montpellier

"Façamos da interrupção um caminho novo. Da queda um passo de dança, do medo uma escada, do sonho uma ponte, da procura um encontro"

Fernando Sabino

Abstract

Many users of multi-agent systems (MAS) are very commonly disinclined to model and simulate using current MAS platforms. More specifically, modeling the dynamics of a system (in particular the agents' behaviors) is very often a challenge to MAS users. This issue is more often observed in the domain of socio-ecological systems (SES), because SES domain experts are rarely programmers. Indeed, the majority of MAS platforms were not conceived taking into consideration domain-experts who are non-programmers. Most current MAS tools are not dedicated to SES, or nor do they possess an easily understandable formalism to represent the behaviors of agents. Moreover, because it is platform-dependent, a model realized in a given MAS platform cannot be properly used on another platform due to incompatibility between MAS platforms. To overcome these limitations, we propose a domain-specific language (DSL) to describe the behaviors of reactive agents, regardless of the MAS platform used for simulation. To achieve this result, we used model-driven engineering (MDE), an approach that provides tools to develop DSLs from a meta-model (abstract syntax), textual editors with syntax highlighting (for the concrete syntax) and code generation capabilities (for source-code generation of a model). As a result, we implemented a language and a textual editor that allow SES domain experts to describe behaviors in three different ways that are close to their natural expression: as equations when they are familiar with these, as a sequence of activities close to natural language or as an activity diagram to represent decisions and a sequence of behaviors using a graphic formalism. To demonstrate interoperability, we also developed code generators targeting two different MAS platforms (Cormas and Netlogo). We tested the code generators by implementing two SES models with the developed DSL. The generated code was targeted to both MAS platforms (Cormas and Netlogo), and successfully simulated in one of them. We conclude that the MDE approach provides adequate tools to develop DSL and code generators to facilitate MAS modeling and simulation by non-programmers. Concerning the DSL developed, although the behavioral aspect of MAS simulation is part of the complexity of modeling in MAS, there are still other essential aspects of model and simulation of MAS that are yet to be explored, such as model initialization and points of view on the model simulated world

Acknowledgements

Tout d'abord, je voudrais remercier le Conseil National de Développement Scientifique et Technologique (CNPq) au Brésil pour le financement de cette thèse et l'opportunité de progresser dans ma carrière scientifique. Je remercie aussi le co-directeur de cette thèse, Pierre Bommel, pour m'avoir invité au sein de l'unité de recherche GREEN, au CIRAD, pour commencer la longue journée qui est une thèse. Puis je voudrais remercier le directeur de cette thèse, Jean-Pierre Müller, pour m'avoir fait confiance malgré les connaissances plutôt légères que j'avais en septembre 2012 sur le système multi-agents. Je lui remercie pour sa patience, ses encouragements, ses indications et pour m'avoir délégué une responsabilité dans un projet ambitieux, où j'ai trouvé mon propre chemin.

Mes remerciements vont également à toute l'unité de recherche GREEN pour l'accueil pendant les quatre ans de thèse. Plus spécialement, à Martine Antona et à Aurélie Botta pour le soutien pendant toute la période de ma thèse, mais également à Jean-François Tourrand et Marie-Gabriele Pikety pour les moments de discussion (et surtout d'amitié) qui m'ont énormément aidé dans les moments critiques de ce travail.

Je remercie Amaury, Paulo, Claudio Almeida, Moises et leurs familles pour l'accueil en France, pour les conseils et pour les moments très conviviaux en famille. Je ne sais comment exprimer ma gratitude à vous tous. De la même façon, merci aux amis Francesca, Antonio, Sylvain et Victor pour leurs encouragements et pour les très bons et agréables moments au CIRAD.

Je remercie les Profs. David Hill et Jean-Michel Bruel qui ont accepté d'être les rapporteurs de cette thèse, et de participer au Jury. Ils ont également contribué par leurs nombreuses remarques et suggestions à améliorer la qualité de ce mémoire, et je leur en suis très reconnaissant.

Enfin, ces remerciements ne seraient pas complets sans mentionner mes amis chercheurs qui ont joué un rôle très spécial pour le début de cette thèse : René Poccard et Marcelo Thales. C'est grâce à eux que j'ai pu commencer cette thèse. Merci à vous.

Contents

List of Figures

List of Tables

List of Codes

List of Abbreviations

MAS	**M**ulti-**A**gent **S**ystems
PM	**P**articipatory **M**odeling
M&S	**M**odel and **S**imulation
SES	**S**ocial-**E**cological **S**ystems
MDA	**M**odel Driven **A**rchitecture
MDD	**M**odel Driven **D**evelopment
MDE	**M**odel Driven **E**ngineering
EMF	**E**clipse Modeling **F**ramework
EMP	**E**clipse Modeling **P**roject
OMG	**O**bject **M**anagement **G**roup
FIPA	**F**oundation for **I**ntelligent **P**hysical **A**gents
MOF	**M**eta **O**bject **F**acility
M2M	**M**odel-to-**M**odel
M2T	**M**odel-to-**T**ext
MOFM2T	**M**OF **M**odel-to-**T**ext
DSL	**D**omain **S**pecific **L**anguage
UML	**U**nified **M**odeling **L**anguage

To my family, for their dedicated partnership and love
in every moment of my life.

Chapter 1

General introduction

1.1 Context

As in any interdisciplinary context, terms that eventually have the same meaning can be differently described by domain experts. Social-ecological systems (SES) are no exception. The domain of SES is an interdisciplinary domain that focuses on understanding and investigating how society relations influence the environment and vice-versa. In this context, Modeling and Simulation (M&S), is one way to understand these inherently complex relations. In (M&S), one must make use of available computer platforms that are able to capture and represent such complexities.

In the past decade, modeling and simulation (M&S) has been used as a promising approach that can give understandings to natural phenomena (Jarrah et al., 2015). Modeling and simulation (M&S) has the ability to increase our understanding of systems, to evaluate them or to predict their evolution (Touraille et al., 2012). At the same time, some Participatory Modeling (PM) approaches have been developed (Andersen et al., 2007; Etienne, 2014; Cardwell et al., 2009), aiming to build a common methodology to design model with stakeholders. One of the key objectives of PM is the attempt to actively involve all stakeholders (e.g. designers, developers, experts, end-users, etc.) in the design process to help ensure that the designed product meets their needs and is indeed usable (Chu et al., 2012).

One of the available approaches in the (M&S) field is the multi-agent systems approach (MAS). MAS allows the modeler to assume the role of an agent during a simulation and thus, to characterize its rules from the ego perspective. Reactivity, autonomy, pro-activity and ability to react to other agents are among the main characteristics that define an agent (see (Ferber, 1999) for a broader definition), where reactivity is specified by set of action-state rules (Bandini et al.,

2009). Reactive agents simply retrieve a pre-set of behaviors similar to reflexes, without maintaining any internal state. Thus, actions contained in that type of behavior can be easily programmed.

Since MAS platforms have been increasingly used to deal with ecological and socioeconomic issues (Promburom, 2002), combining them with PM approaches might be a very efficient method for modeling and discussing social environmental systems. However, even if most of the existing MAS tools share the common basic concepts about what an agent is (role, interaction, reactivity, etc.), current modeling languages differ in the approach of how those concepts are modeled. Consequently, modeling in most of these tools requires stakeholders to adapt to a specific platform philosophy or programing language.

Although the reactivity of an agent can be relatively easy to program, using concepts (i.e. programing language concepts) that are not related to what participants desire to model is one of the reasons why involving them inthe modeling process is not an easy task. Another reason is that most stakeholders are not programmers and thus, may be reluctant to spend much time on the project. Sometimes they are forced into a predefined top-down procedure, or the tools and models at their disposal are fitted with pre-fabricated black boxes that they cannot understand and assess (Ramsey, 2009).Consequently, the absence of intuitive tools using identical vocabulary to that used by stakeholders, may have contributed to their lack of interest in M&S.

1.2 Objectives

Our objective in this thesis is to investigate how new technologies and computer modeling approaches can tackle the problems previously mentioned. More specifically, our focus will be the abstraction of some MAS models in SES domain and platform coding details.

Our first goal is to create a structure of syntactic terms based on the observation of common terms that reflect stakeholder's vocabulary to describe behaviors. To this end, we should consider how behaviors are specified before they are modeled. This specification should contain terms and relations that are more familiar to stakeholders. After identifying such terms, we should be able to construct an abstract syntax and use it to develop a programing language (concrete syntax) that will be as close as possible to the vocabulary used to specify behavior in SES models.

Our second goal is to develop a high-level abstraction language that allows practitioners to focus on modeling without worrying about the simulation aspect. This language should offer enough expressibility to make use of domain terms stakeholders are more used to. As in any language, formalisms should be respected and for that reason, validation syntax rules should be implemented to ensure model validity. Additionally, since the simulation aspect should not be of concern to stakeholders, neither should the specificities of MAS platforms (such as initialization, agents activation and programming language). To solve this issue, we should provide code generators for any specific MAS platform.

1.3 Outline

This thesis is structured into five main chapters. Chapter 2 provides an overview of Modeling and simulation behavior of multi-agent systems. The first sections focus on how behaviors can be specified in MAS, starting with a brief section of what is our vision of a computer agent. The next sections are dedicated to MAS tools used in PM for modeling behavior, followed by an analysis of some of these tools limitations, and possible improvements that could be done in these tools.

Chapter 3 focus on Model-Driven Engineering (MDE) software development methodology. We begin with some background theory and explanation of some terms and definitions commonly used by the MDE and some initiatives that implement MDE. Then, we dedicate the rest of the chapter to the description of the MDE modeling framework we actually used for our works, namely the Eclipse Modeling Project. We show how Eclipse Modeling Framework tools could be used to develop domain-specific languages.

In Chapter 4, we introduce an example of MAS model in the domain of biology. To explain this model, we focus on how agents and their behaviors are usually described by non-programmers. Chapter 5 is dedicated to the definition of an abstract syntax based on stakeholders' specification that was captured by a participatory methodology. We describe a meta-model that was conceived based on terms existing in the SES model introduced in Chapter 4. This meta-model is used as an abstract syntax and, in further sections, we explain the semantic domain and propose a concrete syntax from the conceived meta-model.

Chapter 6 demonstrates the application of Eclipse Modeling Framework to develop a textual editor for a domain-specific language in order to model and

initialize reactive behaviors on MAS. The framework is also used to implement code generators for two specific MAS platforms. The code generators are tested by using 2 SES models implemented with the language proposed in Chapter 5. Later, the generated code is analyzed and used in MAS simulations. Finally, we conclude this thesis in Chapter 7, where we discuss the main contributions exposed in this document, along with some avenues for future works.

Chapter 2

Modeling and simulation behavior of multi-agent systems

2.1 Introduction

In the domain of artificial intelligence, an intelligent agent can be defined as a computational autonomous entity with perception, reactivity and pro-activity abilities (Wooldridge et al., 1995). However, several authors differ in their point of views on these concepts (Franklin and Graesser, 1997) and we still are lacking a consensual definition (Dent, 2007)

A broadly accepted definition for intelligent agents, however, is that of an encapsulated computer system that is situated in a certain environment and capable of autonomous actions and interactions capabilities (Wooldridge, 2009). An autonomous action can be seen as the ability of an agent to decide for themselves in order to satisfy their design objectives. Interaction capabilities can be understood as the ability to mimic our social everyday behaviors, such as cooperation, coordination and negotiation. In this sense, Multi-Agent Systems (MAS) arises from the idea of systems adopting an agent-oriented view of the world, which involves multiple agents and the relationships between them

In this chapter, we discuss how their autonomous behaviors can be expressed, while we explain the most common types of behaviors present in the majority of MAS. Our goal is also to provide an overview of some PM experiences using MAS platforms and how those platforms were used to express agent behaviors. Finally, we will present some challenges linked to modeling with stakeholders using current MAS tools, some limitations of the previously presented tools, and possible improvements to be applied to those tools.

2.2 Behavior and MAS

Intelligent agents can be classified from many different perspectives, according to their intelligence degree (Russell et al., 1995): reflex agents, goal-based agents, utility-based agents and learning agents; or according to their functionalities (Hostler et al., 2005): collaborative agents, reactive agents, mobile agents, interface agents, etc. But those classifications do not focus on the behavioral aspect of an agent. In (Nilsson, 1998), this particular aspect is defined as the process of mapping perceptions to actions. Yet, according to the author, this process can be broken down into the following sequential steps: Sensor data, Perception, Cognition, Reasoning, Goal-setting, Evaluation, Action validation, Action Performance and Learning. Without exhaustively discussing each one of these steps, we provide an example that illustrates the relation between these steps, represented in Figure 2.1.

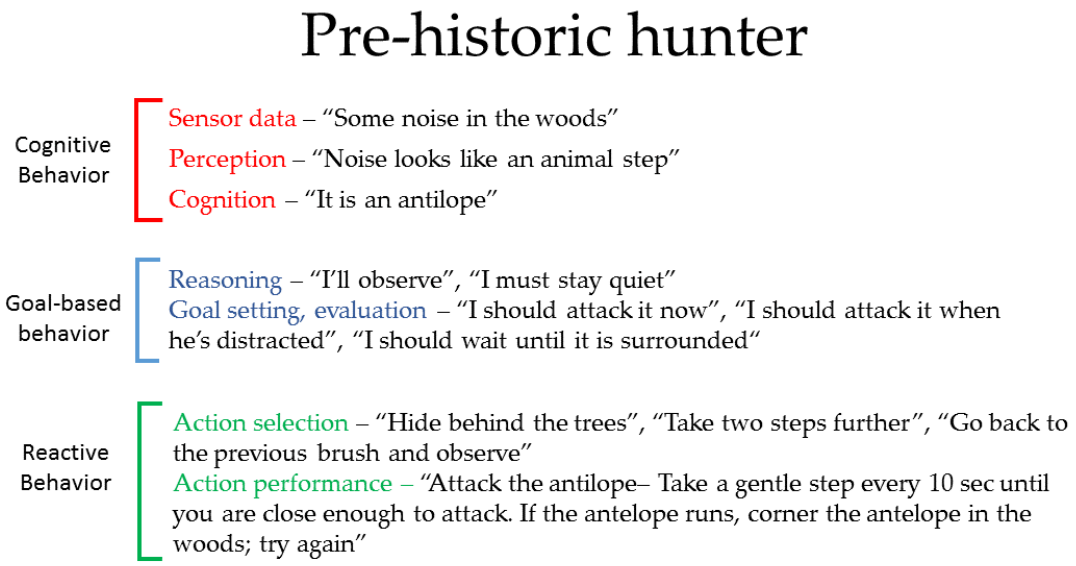


Figure 2.1 – A human behavior process : from cognitive to reactive behaviors

In figure 2.1, a prehistoric human hunting process is used as an example of behavior process. The whole hunting process as a behavior, is divided in many steps. Even if the main behavior is to attack an animal, the attack itself is neither the first nor the last behavior in the whole hunting process. Some steps, such as sensing and perception, are fundamental for goal definition. In the example illustrated in Figure 2.1, depending of the noise or hunting environment, the hunter may decide to wait, or follow the prey a little while more. Defining a goal or a strategy is also part of the hunting process. They serve as parameters for the final action to be taken by the hunter: attacking the prey. Note that

attacking the prey may also involve a set of actions: attacking on the left, on the right, run and attack, attack and run, wait, move left, surround the prey and attack, etc. These sets of actions can be considered as reactive behaviors.

The learning process (although out of the scope of this work) is also worth to mention, since is the last (and a fundamental) part of any intelligence behavior. Learning behavior is the capacity of acquiring experience for further decision making. Or as defined by (Michell, 1997), "*a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E*". As a subfield of artificial intelligence, machine learning makes use of several available techniques (genetic algorithms, neural networks and many others) that are applied to MAS learning capacity.

Since the focus of this work is the behavioral aspect of MAS, we classify the agent's behavior into 3 types of behaviors: cognitive behavior, goal-based behavior and reflexive behavior. These groups were defined based on more detailed definitions of agents behavior made by (Russell et al., 1995), and a classification proposed by (Demazeau and Müller, 1990).

2.2.1 Cognitive behavior

Cognition is the mental action or process of acquiring knowledge and understanding through experience, and the senses[1]. Cognition is largely studied in the fields of linguistics, neuro-science, psychiatry, biology, computer science and many others. Cognition processes are studied by cognitive science, an inter-disciplinary scientific field that seeks to understand how cognitive mental processes work. More precisely, it tries to understand how conscious mental behavior, such as the behavior of thinking, understanding, learning, and remembering, are processed in the human brain. Nevertheless, this process can be more or less complex according to the type of intelligence we want to mimic. In the simplistic scheme in Figure 2.2, the number of steps required to take an action is directly proportional to the complexity of the individual taking this action. In this case, humans may be able to process a higher number of steps before displaying any type of behavior, by contrast with simple organisms found in nature.

[1] "Cognition". Oxford English Dictionary Third Edition, 2010.

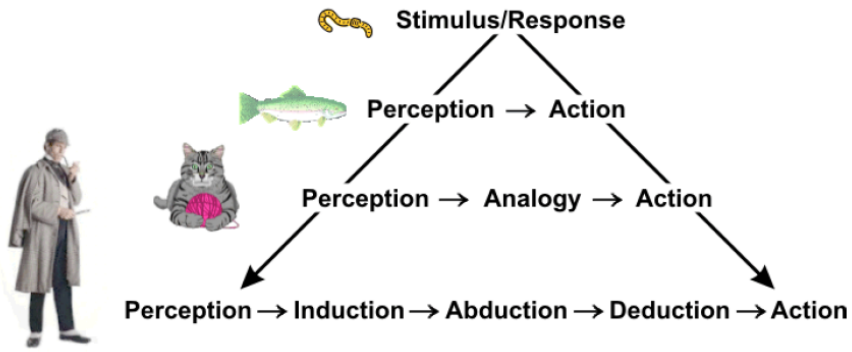


Figure 2.2 – A cognition process.Source : Sowa, 2011

One of the main characteristics of an intelligent agent is their capacity of autonomous actions in the environment, in order to meet their objectives (Wooldridge et al., 1995). But the ability to perceive that environment is also part of a cognitive behavior of any intelligent agent. In that sense, as an interdisciplinary study of philosophy, psychology, artificial intelligence, etc, cognition science played a major role in defining cognition architectures for the perception of intelligent agents. One example is the ACT-R theory (short of "**A**daptive **C**ontrol of **T**hought **R**ational") proposed by the psychologist John Anderson. In the work titled "The Architecture of Cognition" (J. Anderson, 1996), Anderson uses ACT-R theory (J. R. Anderson, 1983) to develop a formal architecture that provides capabilities to simulate cognition.

Many other cognitive architectures were also derived from ACT-R theory, such as SOAR (Newell, 1992), CLARION (Ron, 2006), EPIC (Kieras et al., 1997), ADAPT (Benjamin et al., 2001), and others. The aim of such architectures is to provide a common computer programming architecture that allows, based on ACT-R theory concepts, to specify knowledge-intensive reasoning, reactive execution, hierarchical reasoning, planning, and learning from experience.

2.2.2 Goal-based behavior

A goal-based behavior comes from the idea of a goal: to construct a plan to change current (or given) world state into a desired world state. Those states area analogous to states contained finite state machines (FSM). In finite state machines, states change from one to another when a required (or a set of) condition(s) is detected. In A.I, FSM can be used by agents as a representation of how their states change when a certain event is perceived on the agent's environment or when it is triggered by another agent.

A simple example of FSM would be an automatic door: a door has a state "open" and "closed" that changes between one to another whenever the door sensor detects the presence ("open") or absence ("close") of someone close enough to it. But instead of focusing on how states change, defining a goal-based behavior is instead to define what is the goal of an agent. In other words, based on a set of available and simple tasks, a goal-based behavior does not focus on how those simple tasks are implemented. Rather, a goal-based behavior specifies situations that are desirable.

Consider, for instance, a simplified behavior tree example given in Figure 2.3. In this example, a hypothetical agent has one single goal: to satisfy their hunger. Given a set of actions (on the left of Figure 2.3), and some pre-conditions to each activity (on the middle of Figure 2.3), a behavior tree is specified in order to achieve a goal. In the example, action "pick up the telephone" has, as an effect, the "telephone at hand" state. The telephone, however, can be at close range (in this case, the agent would just have to pick it up) or the telephone can be far (he would have to move), or not in the agent's range (triggering another action to search for the telephone). In any way, the goal "telephone at hand" would be achieved, but in more or less time. Depending on the weight attributed to some specific actions, the slowest option might also be the best one.

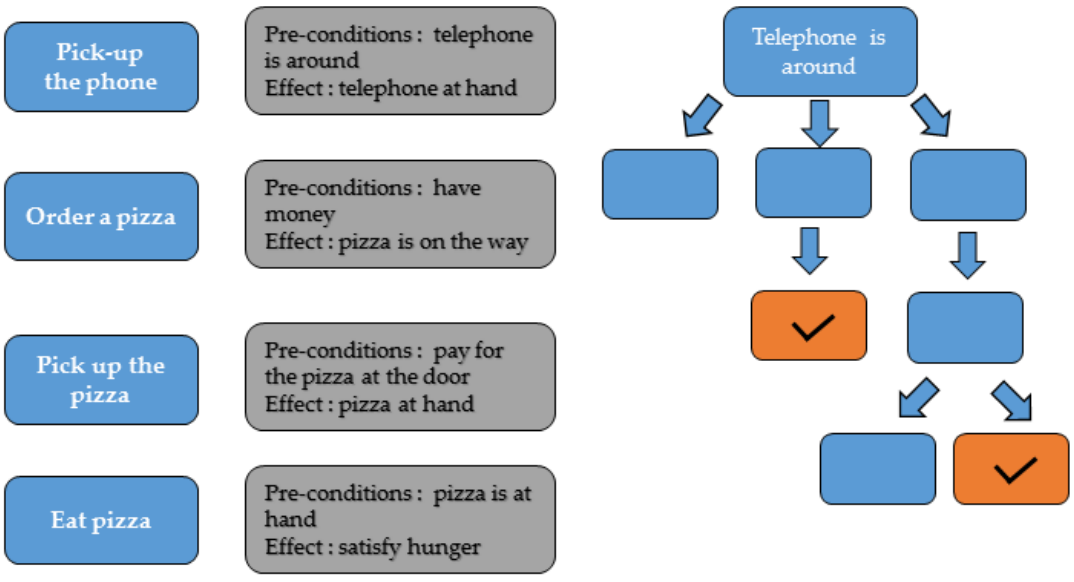


Figure 2.3 – An example of goal based behavior "satisfy hunger" and the many activities that might be executed by the intelligent agent to achieve his goal

On one hand, if a more refined behavior is specified, goal-based behavior may have multiple goals and this could lead to a non-deterministic behavior. But

incorporating goal-based behaviors to MAS can greatly enhance the intelligence of agents.

2.2.3 Reactive behavior

Also termed Simple Reflex Behavior by (Nilsson, 1998), the basic notion of reactive behaviors was first defined in the field of behaviorist psychology by John Broadus Watson, in 1925. By using the notion of "S-R" (Stimulus-Reaction) scheme (see Figure 2.2), any a priori "reasoning" is excluded between S and R, where S is considered as a particular perception of the environment containing the entity, and R as a sequence of basic actions.

One key difference between reactive behaviors and the previously discussed type of behaviors is that in the first, actions are based on the agents current perception or the environment, and not on past perceptions. For example, consider a robot that would be required to gather a specific type of soil sample (peaty soil) from a specific place. Considering that the action of gathering a soil sample is a simple reactive behavior, if the robot finds the same soil sample in a different place, it would then still gather new samples. The robot does not take into account that it has already collected samples. That idea is illustrated through an FSM in Figure 2.4.

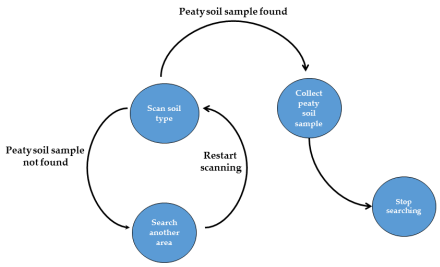


Figure 2.4 – A finite state machine representing a reactive behavior

The reactivity showed in the example above is quite straightforward. It is very similar to a human reactive behavior (e.g. when we dodge something approaching us at a very high speed).

2.3 Participatory Modeling and MAS

2.3.1 Background

In participatory modeling, Role Playing Games (RPG) are very often used as an interactive and collaborative storytelling approach to simulate models. Some approaches successively used RPG on PM for discussion with stakeholders (see D'Aquino et al., 2002 for a list of 5 PM experiments using RPG). Combining RPG with M&S allowed modelers to use RPG as an observation and data collection tool to develop MAS simulation models. It allows individuals to play an active part in their participation, by concretely set out the issues (Gourmelon et al., 2013). But even if that type of approach is very effective for model discussion, in the end the model is almost entirely developed by researchers.

This could lead one to believe that RPG are enough to simulate models, and MAS would not be of much use. Yet, as stated by (Barreteau et al., 2000) two points must be considered. One is that RPG alone as a modeling tool are reported to be limited as they are cumbersome and slow to develop, and analysis of their results is still difficult. Another point is that comparison between different experiments using RPG is difficult since many parameters in a game are not controlled.

To overcome these issues, computer games based on repeated RPG observations were developed. That is the case of FishBanks (Meadows et al., 2015),a multi-player web-based simulation in which participants play the role of fishermen and seek to maximize their income as they compete against other players and deal with variations in fish stocks and in their catches.

One of RPG concepts is that, by taking the part of a character in the game, the player must follow rules that have been previously defined by the game creator. This affords, however, little flexibility to players during a game, since rules cannot be changed. In real case scenarios, such as natural resources management for example, discussions can last hours due to natural conflict of interest between stakeholders. Also, the addition of new rules or new parameters might be necessary. Rules flexibility of should be considered if we wish to capture how stakeholders mediate and solve their conflicts. This understanding is essential to capture the ways they behave.

The fact is that, in most cases, scientists are inclined to favor the tools that they are most familiar and comfortable with (Voinov and Francois Bousquet, 2010).

As a result, some works (as described in section 2.4) tried to incorporate new tools into MAS platforms. Their aim was to add new visual or textual languages to facilitate the M&S process and consequently, behavior modeling with stakeholders. Some of these experiences are presented in the next section.

2.4 Experiences of MAS tools used in participatory modeling

2.4.1 CORMAS

From **CO**mmon-pool **R**esources and **M**ulti-**A**gent **S**imulations, Cormas (François Bousquet et al., 1998) is a MAS tool that focuses on models for renewable natural resource management. According to the authors, CORMAS is oriented towards the representation of interactions between stakeholders about the use of renewable natural resources. Based on the VisualWorks programming environment, users can develop their model using SmallTalk programming language.

Used in several PM works[2], CORMAS provides some facilities to interpret (and modify) the behavior of agents during M&S. An executable activity diagram (Figure 2.5) allows to directly change the behavior of agents through a UML activity diagram. The diagram is, in turn, interpreted by the agents and can be modified by the users to redefine the order on which behaviors will be executed.

The diagram was used in ((Bommel, Dieguez, et al., 2014, and Bommel and Dieguez, 2011)).The authors stated that, although the aim of the tool was not to generate the entire simulator code, it was able to improve the collective modification of a model. Moreover, the tool was based on a simplified version of UML's activity diagram. Since a tool based on a full specification of UML's activity diagram could discourage potential users, a simpler version enabled anyone involved in the modeling process to participate more actively.

[2] An exaustive list models (but not all involving PM) can be found at : http://cormas.cirad.fr/en/applica/tousmodeles.htm

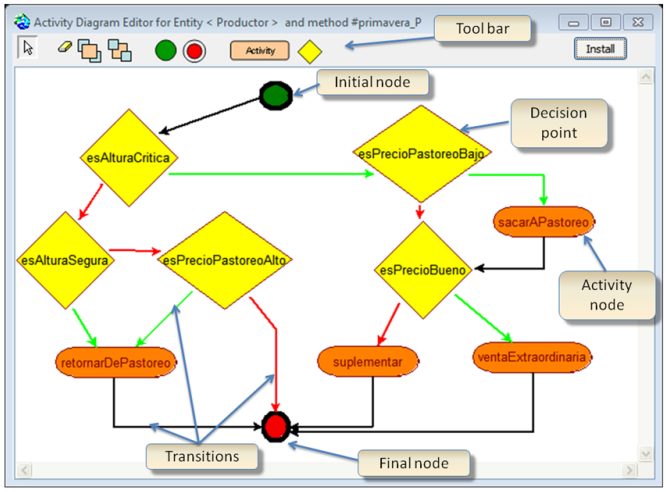


Figure 2.5 – Cormas activity diagram tool to interpret agent's behavior. Source : Bommel and Dieguez, 2011

2.4.2 NetLogo

NetLogo (Wilensky, 1999) is a multi-agent programmable modeling environment. Netlogo allows quick development and prototyping from simple to complex models. That is due to the fact Netlogo programming language is based on the Logo language, a dialect of Lisp that was designed for learning and other educational purposes. As a result, Netlogo's data structures (such as words and lists) closely parallel the words, phrases, and sentences that make up spoken and written language. Plus, logo-based languages provide the user with a good feedback on individual instructions, helping in the debugging and learning process.

With many applications in a plethora of domains[3], agents in Netlogo are programmed in the form of turtles, patches, links and the observer. The most cited MAS tool in the last decade (Page et al., 2012), NetLogo uses grids, bars and charts to represent agent's environment and interactions. It also offers a System Dynamics Modeler, which allows the user to draw diagrams that define populations of agents as "stocks" and flows.

In a PM experience concerning contagious human diseases (Maharaj et al., 2011), the authors developed a graphical user interface (GUI) for Netlogo to be used in participatory experiments aiming at investigating human attitudes toward the risk of being infected by a disease.

[3]Netlogo model library, also available in the version for download: http://ccl.northwestern.edu/netlogo/models/

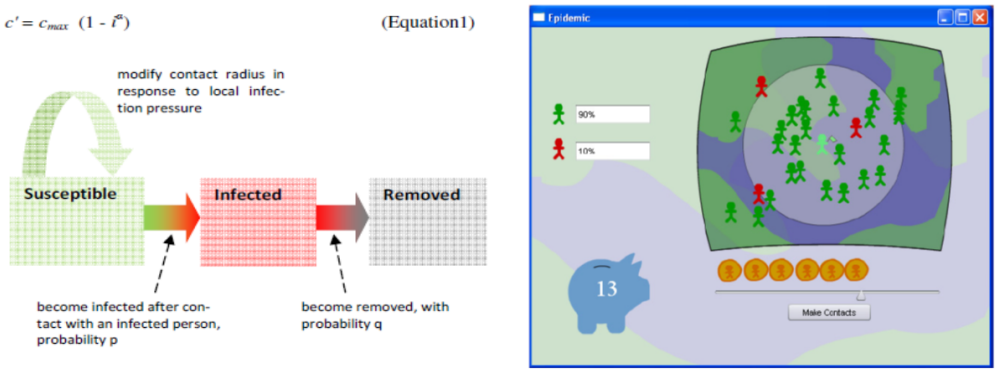


Figure 2.6 – Netlogo in a participatory modeling experience for human epidemiological study. Source : Maharaj et al., 2011

2.4.3 Anylogic

AnyLogic (Borshchev, 2007) is a multi-paradigm modeling platform based on Java that provides capabilities to develop and combine agent-based models, system dynamics models, discrete-event models, and continuous and dynamic system models. Although AnyLogic does not focus on SES and though it is not under GNU GPL license, the platform has been used in some participatory modeling on SES domain.

In (Gaube et al., 2006) for example, AnyLogic was used for modeling the impacts of subsidy policy on farmer households, land use and nutrient flows. In this work, the behavior of the agents could be affected by the system they are part of and by changes in their environment (in this case, land use). Consequently, the behavior of the whole system would depend on the individual behavior of each agent. The model was programmed in Java and later used in participatory processes.

Another example of Anylogic's usage in the context of SES can be found in (Tàbara et al., 2007), where the authors aimed at using participatory modeling for integrated water sustainability assessment. Interestingly, the authors implemented a first version on Netlogo to describe the physical system (that is, the hydrology system) and the agent system that describes the behavior of different agents. Then, AnyLogic was applied to develop an agent model that represented farmers (for agricultural water), electricians (for water dependency), frogs (water environmental use) and households (human water consumption besides agriculture). The agents could interact with and be influenced by their surroundings and the environment (the physical system) whereas the behavior of and

decisions made by the agents were in turn reflected in the physical model. Finally, certain initial parameters for the agents' behaviors were defined by the players and assigned to the agents, as represented in (2.7)

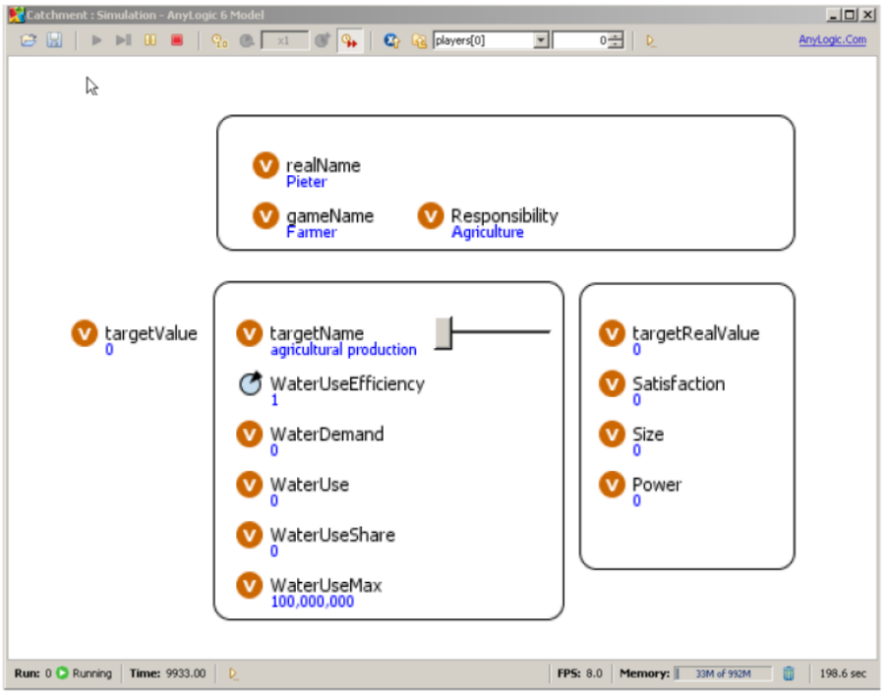


Figure 2.7 – Agents' behavior parametrization on Anylogic. Source : Tàbara et al., 2007

2.4.4 GAMA

The GAMA platform (Taillandier et al., 2012) - short for **G**IS & **A**gent-based **M**odeling **A**rchitecture, is a modeling and simulation development environment for building spatially explicit agent-based models. GAMA has its own modeling language (GAML - GAma Modeling Language) and it was based on XML. The GAML language focus is simplicity and considered by the authors to be as simple and easy to understand as the Netlogo modeling language (Taillandier, 2014). Gama has a declarative user interface, focusing on GIS and multi-layer 2D/3D visualization.

Although GAMA possesses many visualization options and a programming language focused on simplicity (GAML), few works relate that platform to PM experience. However, in (Chu et al., 2012) GAMA was applied in an approach proposal based on a participatory design to find the most effective way to model human behavior. As a part of a broader PM methodology, a participatory design attempts to actively involve all stakeholders (e.g. designers, developers,

experts, end-users, etc.) in the design process to help ensure that the designed product meets their needs and is usable.

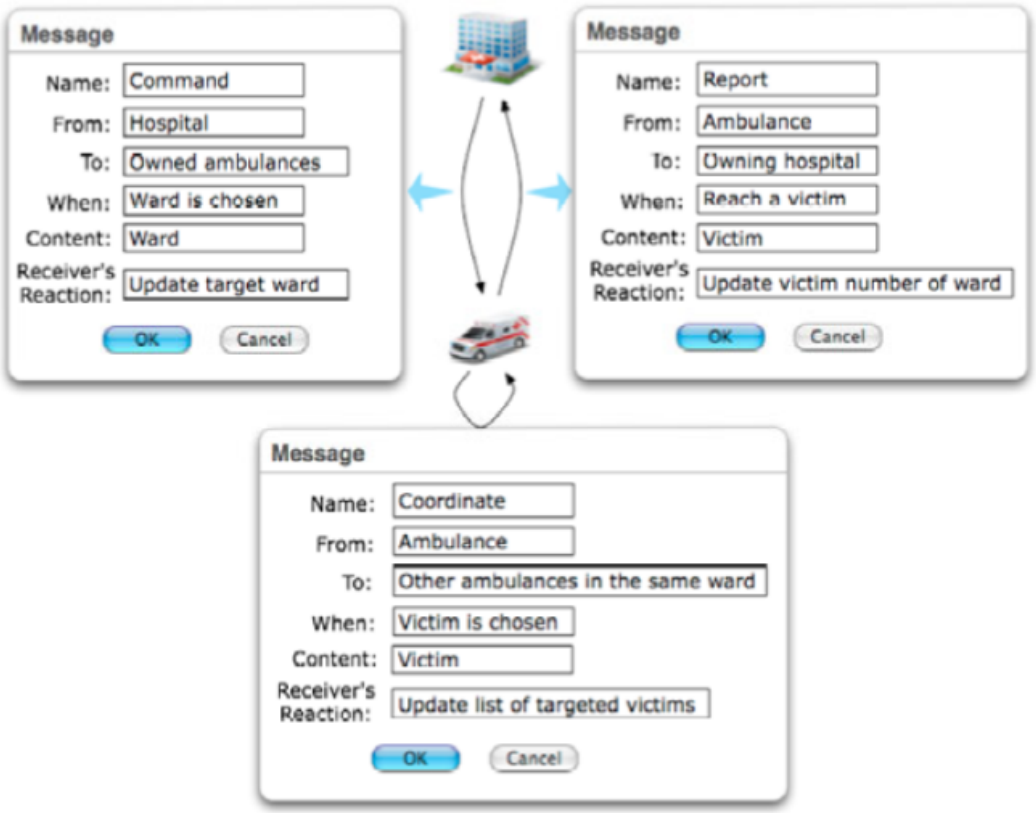


Figure 2.8 – GAMA and the interaction view in the urban emergency management context Source : Chu et al., 2012

In this work, the authors developed a model in the context of urban emergency management, aiming at the organization of resources and responsibilities for dealing with many aspects of emergencies occurring in cities (e.g. search and rescue of injured persons, extinguishing fires, etc).Among many available "views" (parameter view, chart view , display view, agent view, etc), GAMA provides the interaction view (Figure 2.8), which shows the coordination occurring between the agents and allows the users to add or modify the messages they are exchanging. The messages, however, are modeled with GaML.

2.4.5 MIMOSA

MIMOSA[4] is a MAS platform aiming at providing tools capable of managing a process from building conceptual models to running the simulations. Through a visual ontology editor, MIMOSA allows the description of the concepts, the structure and the relationship of agents to be made through ontologies. The modeled ontology can be used together with an extensible set of formalisms for model initialization and model visualization. Additionally, the conceptual model can be designed using a simplified UML class diagram. For the model dynamics (like agent's behaviors), Mimosa allows the user to choose the formalism to be used, conditioning the possible states, and initialize model dynamics. Mimosa's simulation kernel is based on DEVS (Zeigler and Sarjoughian, 2003)

In (Aubert, Müller, and Ralihalizara, 2010), MIMOSA platform was used to develop an agent-based model called MIRANA. Mirana model aimed to evaluate the impacts of the management plans made by Malagasy local communities and explore the impacts of their decisions in scenarios. In this work, the conceptual model was made of a set of ontologies describing the actors of the system (households, communities, etc.), objects on which they were acting on (lands, animal and plant species, etc.), actions carried out by the actors on the objects (hunting, cropping, etc.) and the regulations on the actions. The actors are provided with needs (food, money, etc.) or objectives (conservation, production, etc.) and planning mechanisms. MIRANA model was composed of two kinds of dynamics: the first one was the biophysical dynamics, representing the population growth of the species and the evolution of the fertility expressed and represented as equations of time; the second one was the decision process of the agents, representing the households and the collective actors such as the local community.

In a more recent work (Aubert and Müller, 2013), the authors improved the MIRANA model by incorporating normative rules to models. An ontology was modeled considering definitions (and relations between) institutions, stakeholders, resources and territories. In the modeled ontology, the institutions and stakeholders respectively represented the "macro" and "micro" levels of description. They considered institutions as a set of constitutive and regulative norms, where organization is a representation of a concrete group of people on whom the institution applies (e.g. household). Stakeholders were considered to

[4]From the french acronym for "**M**éthodes **I**nformatiques de **MO**délisation et **S**imulation **A**gents": computer science methods for agent-based modeling and simulation

be exclusively active, human, individual or collective, decision-making parties with objectives. To represent the dynamics between MIRANA model elements, the authors used a mixture of natural language descriptions and UML activity diagrams. An example of dynamics representation is illustrated by Figure 2.9 where household behavior is conceptualized in an UML activity diagram

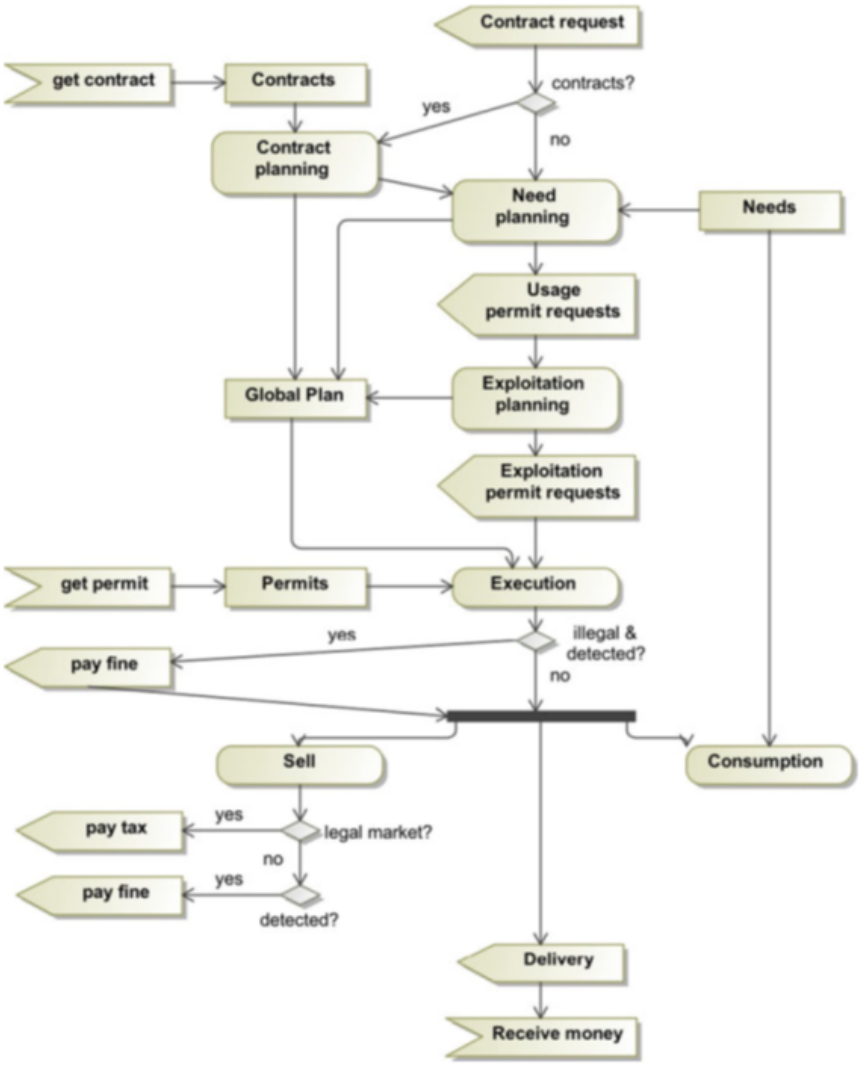


Figure 2.9 – Mirana and Household behavior. Source: Aubert and Müller, 2013

In MIMOSA, the modeler has to define two types of models: the conceptual model (that corresponds to the concepts and their relationships) and the concrete model (that corresponds the instantiation of the conceptual model). Those descriptions are further used to generate a simulation model in the form of DEVS, and the whole model, written in Java.

2.5 Limits of the current MAS tools used in participatory modeling

2.5.1 The Challenge of MAS behavior simulation in SES

In M&S, a formalism must be chosen if one desires to simulate a computer model. Concerning SES, the chosen formalism should be able to provide most of the concepts involved in SES elements, such as territory, environment and actors. These concepts are essential for MAS modeler behavior, especially if we consider that the way behaviors are modeled is strongly influenced by the way these concepts are formalized.

In MAS, the concept of environment is essential for M&S of agent-based models. Unfortunately, the multiplicity of disciplines involved in social environmental systems (SES) results in a non-consensus about many of the concepts involved in SES. To exemplify this, in his work titled "The Significance of Territory" (Elden, 2013), the author discusses how the concept of territory may vary from group to group. Not only do the concepts vary, but interests are also very different. For instance, politicians see the territory as population aiming at resources while militaries view the territory as topographic features aiming at tactical and strategic actions. To jurists, the territory is a jurisdiction and delimitation since it aims at national and international laws. To the geographer, it could be a portion of space enclosed by boundary lines, but one interested in political geography might see the territory as a material, spatial notion establishing essential links between politics, people, and the natural setting.

Because of this diversity in disciplines involved in PM, it is still not very common for stakeholders to find the concepts that they are used to in the current MAS platforms. Covering concepts from all disciplines does not seem to be a logical solution, even though a consensus on these concepts would be ideal. Providing stakeholders with a way of expressing their own concepts in a formalism they could easily understand is a task still to be accomplished in MAS.

2.5.1.1 Programming language learning

One of the main obstacles for stakeholders during M&S phase is learning a new programming language. They tend to be discouraged during this stage because

a majority of stakeholders are not programmers. Additionally, much of stakeholders time during PM is consumed during sessions of discussing and conceptualizing a model. Furthermore, even non-programmer researchers might spend quite some time learning and applying a programming language to specify behaviors in a given MAS platform.

In a PM experience concerning contagious human diseases for example (Maharaj et al., 2011), the authors concluded that, although the tool was created to be interesting and engaging for participants, it is still very far from a real-world experience of an epidemic reality. One of the reasons might be due to the fact that behaviors specified in the model were predefined by developers. Also, some MAS platforms (such as AnyLogic), "*...have their use still confidential and regarded as "too advanced" or "too complicated" by many adaptation planning projects in PM* (Drogoul, 2015). Although there are more user-friendly platforms, and although enhancing the education and training of stakeholders and deciders could help overcome this issue, stakeholders still spend quite some time on training and learning programming languages, if they wish to simulate their model.

2.5.1.2 MAS behavior representation

Concerning the behavior aspect of MAS tools in most PM experiences, the behavior represented is usually a reactive one. One possible reason is that the implementation of cognition and goal-based behavior could be very difficult to program, especially for non-programmers. Although the idea of following cognition and goal-based architectures would highly increase the detail level of a model description, following these architectures would proportionally increase the complexity of representing that model. Certainly, this does not mean, though, that cognition and goal concepts are not present in MAS tools, but rather, that cognition and goal-based architectures should be implicitly incorporated by developers of MAS platforms that are commonly used by stakeholders.

In general, there are three ways to graphically represent behaviors in MAS simulation: through system dynamics, through DEVS, or through activity diagrams. The last can be considered as a good ratio of intuitiveness/level of formalization since it is also suitable for representing reactive behaviors due to its control flow nature. MIMOSA and CORMAS advocate the need for conceptual modeling before simulating models, providing an Ontology and an Activity diagram editor, respectively, as a way to facilitate modeling.

Nonetheless, class diagrams and ontologies are only capable of describing the general structure of the model and relations between them. Which means that agents' behaviors are always hand-coded by researchers. In the recent attempt (session 2.4.1) to increase stakeholders involvement during M&S process, the activity diagram editor only allowed users to visually modify the ranking of behaviors that were previously programmed by an expert modeler. The authors also stated that the tool does not prevent the modeler from programming his ABM.

Finally, the chosen formalism to represent a behavior should not be exclusively coupled with the programming language paradigm of the chosen MAS platform. For example, if Netlogo and Anylogic provide many ways to model behaviors (like slide bars, text, stack and flow, etc), CORMAS on the other hand uses UML diagrams because, among other reasons, UML was designed to visually represent models according to the object-oriented paradigm, and because CORMAS programming language (SmallTalk) is also object-oriented.

2.5.2 Possible improvements to current MAS tools

2.5.2.1 Domain specific language for MAS behavior modeling

Most MAS platforms presented so far are based on programming languages that were not designed through taking into account concepts present in a stakeholders' discourse. Correspondingly, stakeholders are forced to deal with programming language aspects and MAS platform specificities. To solve that issue, "what" a model should simulate must be separated from "how" the same model should be implemented. In other words, the simulation model must be separated from the conceptual model. This could be achieved by implementing an intermediary layer, significantly reducing M&S complexity for non-programmers. This intermediary layer could be represented by another programming language, composed of concepts that are common to modelers and stakeholders.

Many aspects (such as type system, implementation, etc.) are used to distinguish and classify programming languages. Programing languages may also be distinguished by their type: they can be general-purpose languages (GPL), or domain-specific languages (DSL). GPLs are programming languages designed to be used for writing software in a wide variety of application domains. Java, Smalltalk, C++, are some examples of GPL. DSL on the other hand, can abstract

the design and implementation expertise of a domain: the DSL programmer focuses on what to compute or to describe, as opposed to how instructions should be computed. Even if DSLs are less understandable for those who do not belong to the targeted language domain, they are much more expressive in their domain, and consequently, a DSL exhibits minimal redundancy. Some examples of DSL include HTML for web markup, SQL for relational database and Mathematica for symbolic mathematical computation.

Because domain-specific languages are more compact (Mernik et al., 2005; Fowler, 2010) and typically far less powerful than generic programming languages, they communicate their intent far better. This is because DSLs do not include many features found in general-purpose programming languages. While it is intended to increase expressiveness (thus decreasing maintenance cost), GPL contains features that also require a certain amount of domain and programming expertise when they are used. But stakeholders are mostly non-programmers, so the use of GPL may discourage them if they wish to simulate their model. DSLs, on the other hand, could highly increase stakeholders' participation level by incorporating some of their domain's concepts in current MAS programming languages and tools (see Mernik et al., 2005 for an analysis of when and why DSLs should be developed and Kosar et al., 2010 for an empirical comparison between DSLs and GPLs).

With the exception of Netlogo and GAML, most programming languages used in the MAS platforms presented are GPL. However, neither Netlogo, nor GAML were specifically made for SES. But even if they can be used, and since social environmental systems are very often modeled using MAS platforms, we still lack DSLs for the SES domain.

2.5.2.2 Platform independence

As intuitive or user-friendly a MAS tool could be, it should be flexible enough to provide the user with tools for quick model validation. One way to validate a model is also to confirm the reproducibility of that model on any platform. But this way of model validation is still not possible in current MAS platforms because models are platform dependent. This means that specific platform elements (such as icons, colors used and the how models are initialized on a specific platform) are modeled along with the model specification. The model should thus be platform-independent.

As an intermediary layer that separates the simulation model from the conceptual model, DSLs may easily provide code generation facilities. One of DSL advantages are the ability to generate simulation models from conceptual models. This is achieved by defining text templates that read a model of the DSL and generate text files. Developers use domain-specific languages to construct models that are specific to their applications and use models to generate source code. Although building DSLs is not a recent approach, and has already been applied to MAS domain (Challenger et al., 2014,Demirkol et al., 2013), DSL code generation capabilities have never been used in PM, specially to describe the behaviors of reactive agents.

2.5.2.3 Visual tools to model with stakeholders

CORMAS and MIMOSA use activity diagrams to model reactive agents behaviors. But MIMOSA experience in PM only used activity diagrams during the conceptual stage of modeling. CORMAS provided stakeholders with a visual tool to "interpret" graphically model behaviors through an activity diagram, meaning that behaviors were previously defined. Additionally, behaviors were hand-coded by expert modelers and not by stakeholders.

Domain-specific languages are usually textual, but can also be visual. In that sense, visual languages are used to communicate a representation of a system, in order to understand or change it (Renger et al., 2008). "Thinking by diagrams" (Blackwell, 2001) affords more efficiency than a text, or a stack of instruction blocks. Also, they seem more powerful than textual description or lines of code, especially for collective design involving non-computer specialists such as local stakeholders (Bommel, Dieguez, et al., 2014). But computer simulations require some kind of formalism to run simulations. So, should a visual formalism be incorporated to a DSL capable of reproducing reactive behaviors in MAS, then that visual formalism should also prove useful in promoting a multidisciplinary discourse among stakeholders (like activity diagrams, for instance).

In PM experiences, flux diagrams (such as activity diagrams) and cause-effect diagrams (such as ARDI[5] (Etienne et al., 2011) are very popular among stakeholders. These types of diagrams can provide a way to represent decisions (represented by agents behaviors on MAS). Moreover, the popularity of these diagrams is due the fact that many policy makers start their professional education

[5] ARDI - From **A**ctions, **R**esources, **D**ynamics and **I**nteractions

by learning how to construct cause-effect diagrams and means-end branching trees (Mayer, 2009). However, cause-effect diagrams poorly describe the spatial and environment aspects of an agent and these types of diagrams might not be enough to represent behaviors.

Nonetheless, a programmer's mindset is still needed when using such visual modeling editors because "users have to deal with concepts such as if-then-else statements, loops, variables usage and so on" (Michel et al., 2011). Despite of that, visual modeling languages have great advantages. They have special virtues such as visualization, immediacy, spatiality, creativity, and compliance with intuition (Morand, 2000).

2.6 Conclusion

In this chapter, we presented how behaviors are represented in some MAS architectures. Participatory modeling usually makes use of MAS tools because MAS is an individual-centered approach. In most PM experiences with MAS tools, reactive behaviors are the most used type of behavior due to their simplicity of programming. Although many advances were made in the past decade, the current MAS platforms are still not able to sufficiently involve participations in M&S simulation process. This is mainly observed in social environmental systems, where most stakeholders are non-programmers. Visual languages to represent reactive behavior, platform dependence and programming languages that are not specifically designed for SES domain are among the reasons why current MAS platforms are still far from keeping stakeholders' level of participation. It is undeniable that much work has been carried out in this field for the past decade, but much remains to be done. Still, we conclude that domain-specific languages should be designed for SES, to which some kind of visual formalism should be incorporated.

Chapter 3

Model driven engineering

3.1 Introduction

So far, we have been using the term "stakeholders" to denote a group of users (including researchers) for whom MAS platforms should be developed. Terms such as "clients" or "users" could also be applied.

Considering that stakeholders (including researchers) are experts in their own domain, the term "domain expert" seems more convenient to use. A domain expert is a person with special knowledge or skills in a particular area of endeavor. The term domain expert is frequently used in expert systems software development, and there the term always refers to the domain other than the software domain. Therefore, the term "domain-expert" will be used instead.

In the domain of software development, a programming language is usually used to develop a software. As previously introduced in chapter 2 programing languages can be a general-purpose language (GPL) or domain-specific languages (DSL). DSL "are small, usually declarative, offering expressive power focused on a particular problem domain" (Van Deursen et al., 2000). Also, MDE has adopted the term 'toolsmith' to refer to developers that use graphical frame-works to build plug-ins. Although many methods for developing DSL are available in the literature, this chapter will focus on Model-Driven Engineering (MDE). We provide a brief overview of MDE, focusing on some of the necessary concepts to develop DSLs. Finally, we explore MDE's perspective on those concepts and how they are implemented.

3.2 Background

Before getting into the "E" of MDE, some "MD" past initiatives should not be neglected if we are aiming at understanding MDE goals. Even before previous model-driven initiatives (as illustrated in figure 3.2),), "various past efforts have created technologies that further elevated the level of abstraction used to develop software" (Schmidt, 2006).

With little impact during the 1980s and 1990s, Computer-aided software engineering (CASE) focused on developing software methods and tools that enabled developers to express their designs in terms of general-purpose graphical programming representations, such as state machines, structure diagrams, and data-flow diagrams. However, CASE tools were unable "to scale to handle complex, production-scale systems in a broad range of application domains" (Schmidt, 2006). Later, developers start to typically use more expressive object-oriented languages, such as C++, Java, or C#. Re-usability was one of the main advantages that allowed such languages to implement complex solutions in mature enough platforms. By using reusable functionality, middlewares (such as J2EE, .NET and the Common Object Request Broker Architecture - CORBA) were created in order to offer solutions to frequently encountered problems on development : heterogeneity, interoperability, security, dependability, etc.

Later, developers started to typically use more expressive object-oriented languages, such as C++, Java, or C#. Re-usability was one of the main advantages that allowed such languages to implement complex solutions in mature enough platforms. By using reusable functionality, middlewares (such as J2EE, .NET and the Common Object Request Broker Architecture - CORBA) were created in order to offer solutions to frequently encountered problems in development: heterogeneity, interoperability, security, dependability, etc.

Meanwhile, with the advances and the increasing popularity of other standards defined by the Object Management Group (OMG), the not-for-profit technology standards consortium decided (around 2001), to adopt a new framework termed the Model Driven Architecture MDA. MDA is in fact, a set of many OMG's standards: the Unified Modeling Language - UML, the Meta Object Facility (MOF), the XML Metadata interchange (XMI) and even CORBA. This idea is depicted in the official logo of MDA (Figure 3.1), illustrating the idea of an architecture composed of many standards, which aims to be used in many sectors in software development. Some of the standards used by MDA will be

discussed in further sections.

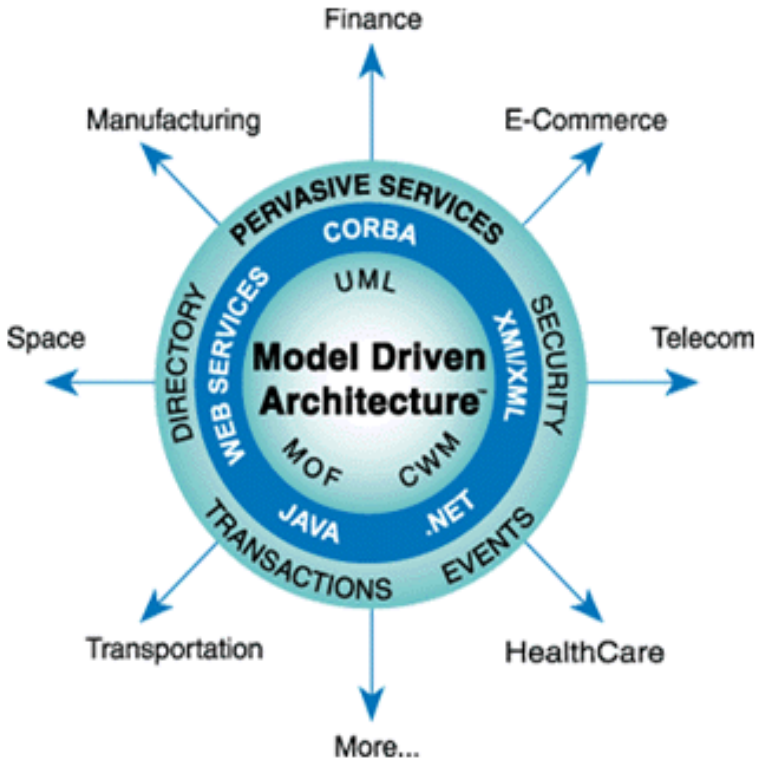


Figure 3.1 – MDA's official logo . Source : OMG, 2014

Nevertheless, unlike other OMG's standards, MDA offers a way to use models instead of the traditional source code (Sacevski and Veseli, 2007). In that sense, the software engineering approach that uses models to create products, increasing quality, efficiently and predictability of large-scale software development is called Model-driven development (Beydeda et al., 2005). Model-Driven Development- MDD (Pastor et al., 2008) is very often called MDSD - Model-Driven Software Development (Stahl et al., 2006) and although there is no consensus on these terms, our understanding is that both approaches have quite the same philosophy: that software development's primary focus and products are models rather than computer programs. The idea is to express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain (Selic, 2003).

Finally, Model Driven Engineering (MDE) has a wider scope than MDD (or MDSD). MDE covers the whole engineering process to develop software (such as tasks, documentation, etc..) by combining it with architectures (Kent, 2002). In short, MDA would be OMG's point view (there may be others) of how MDD (or MDSD) should be implemented. And on the top of hierarchy (depicted by figure 3.2), we have MDE, a software engineering approach. Boosted by MDA,

MDE is very much established in the industry, and is supported by important groups such as IBM, Microsoft and OMG (Touraille et al., 2012). We will discuss MDE further in section 3.7

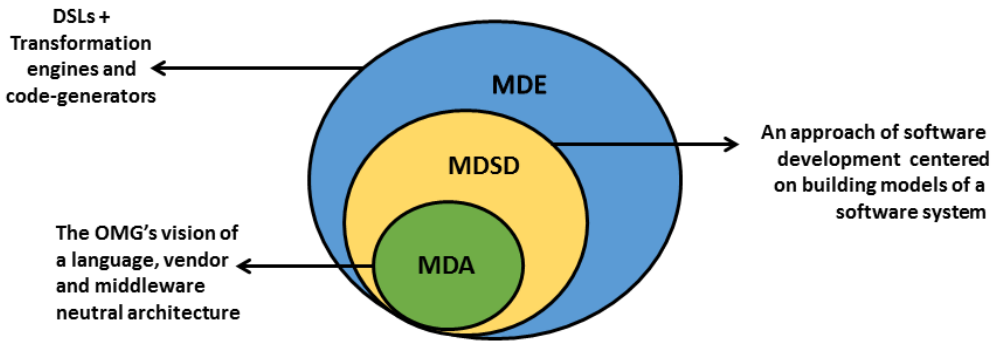


Figure 3.2 – Model-Driven Engineering (Adapted from : Cabot, 2009)

3.3 Model

In its most simple definition, a model is a simplification of reality. In a more broad definition of a model, (Le Moigne, 1990) state that:

> *"A model is an artificial representation that we built in our minds..which we design on some surface object : the beach sand, a leaf of paper, the screen of a computer....also understood as a system of symbols, an artificial system (created by men) that arranges symbols..."*[1]

Models can be physical, such as a toy car or an architect's model of a building, or symbolic, such as a natural language, a computer program, or a set of mathematical equations. Considering certain characteristics and the "Nature of the definition of a model", (Rothenberg et al., 1989) affirm that:

> *"A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibly of reality."*

Accordingly, a model can be a representation of a selected part of the world (the system). A model can represent a theory in the sense that it interprets the

[1] From the original text : "Le modèle est une représentation artificielle que «l'on construit dans sa tête»...et que l'on «dessine» sur quelque support physique : le sable de la plage, la feuille de papier, l'écran du «computeur»... Autrement dit un système de symboles, un système artificiel (créé par l'homme) qui agence des symboles..."

laws and axioms of that theory. These two notions are not mutually exclusive as scientific models can be representations in both senses at the same time (Frigg and Hartmann, 2012). However, because a model may concern different aspects of a system, MDA introduces the notion of viewpoints. Viewpoints in MDA Viewpoints in MDA are "a reusable set of criteria for the construction, selection, and presentation of a portion of the information about a system, addressing particular stakeholder concerns" (OMG, 2014). Therefore, MDA defines 3 viewpoints: 1) a Computation Independent Viewpoint, 2) a Platform Independent Viewpoint, and 3) a Platform Specific Viewpoint. As shown in Figure 3.3, all of these viewpoints are captured in the following models:

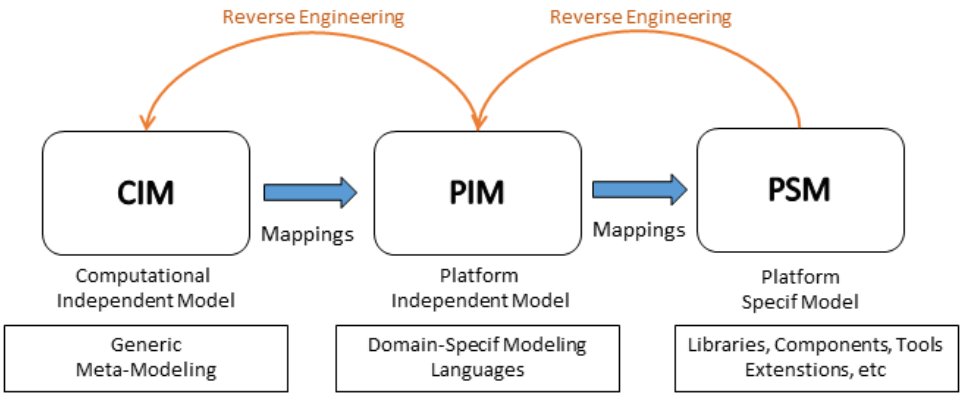


Figure 3.3 – MDA models

- The Computation Independent Model (CIM): is basically a simple representation of a system without specific information on how it is to be implemented. A CIM does not carry any detail of the structure of systems, but it focuses solely on the description of the domain and requirements of the system.
- The Platform Independent Model (PIM): is a somewhat more detailed view of a system containing more details but from a platform independent viewpoint. It captures information on the data of a system and, from the computational viewpoint, leaves aside all lower-level details;
- The Platform Specific Model (PSM): is a view of a system from the platform specific viewpoint. A PSM contains PIM specifications but with details about the usage on a concrete platform. Probably generated from a PIM, the PSM must be precise and complete, conforming to the constraints of a specific (class of) platform(s). The PSM usually contains enough information to allow the generation of codes.

3.4 Modeling Language

Considering it as the representation of a system, a set of theories, or both, a model is a description of (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer (Kleppe et al., 2003). For a model to be useful, OMG, 2014 recommends that:

> *"A model needs to be expressed in a way that communicates information about a system among involved stakeholders that can be correctly interpreted by the stakeholders and supporting technologies. This requires the model to be expressed in a language understood by these stakeholders and their supporting technologies."*

As illustrated in figure 3.4, a modeling language is composed of an abstract syntax (expressed in a meta-model, see 3.5), a concrete syntax (represented by notations) and semantics.

Figure 3.4 – Main elements of a modeling language

While the abstract syntax defines the structure of the modeling language, independent of any particular representation or encoding, concrete syntax represents a particular way to represent that structure. Thus, the concrete syntax must always be conform to an abstract syntax. As the third element of modeling languages, semantics reveals the meaning of syntactically valid expressions of that language (Rodrigues Da Silva, 2015).

For instance, let us suppose that we wish to implement a modeling language for modeling classes in object-oriented languages. Usually, a class is abstractly de-fined by a name, many methods (which in turn has its own abstract syntax) and a modifier. Each object-oriented programming language (such as Java, C++,SmallTalk, etc) have their own (concrete) syntax to implement the abstract

syntax that defines a class. If we hypothetically wish to develop a new programming-oriented- language, we could use the same abstract syntax of previous object-oriented programing languages and provide our own concrete syntax.

In other words, designing a modeling language is the process of setting up all those elements together, that should provide enough expressivity to create models. UML, SQL Schema, are two examples of well-known modeling languages. They have their own semantic modeling domain, such as ontologies (for UML) and data-base (for SQL). From a single abstract syntax, they may have several concrete syntax.

3.5 Meta-model

Designing modeling languages requires a higher level of abstraction. We call that level a meta-model. A meta-model is set of objects that describes the domain model. That means that all modeling languages (such as UML, SQL, OWL, and other) also have their meta-model. In that way, UML, for instance, may be considered both as a model and a meta-model. OMG for example, use a core meta-model (also in UML) to define some of MDA's standards (some of them are dis-cussed in further sections). This core meta-model is called Meta-Object Facility - MOF (OMG, 2015) and it is based on a simplification of UML's class modeling capabilities. MOF is designed as a 4-layered architecture, as illustrated in Figure 3.5.

At the bottom level, the M_0 level represents the system. The M_1 level represents the models of real system in the view point of the uses. These models conform to a meta-model represented by the M_2 level, level, which in turn, conforms to a meta-meta-model defined in M_3 level. In the the MOF's M_3 viewpoint level, a meta-meta-model should be able to represent itself. The UML formalism, for example, is a formalism that can be represented by itself. It also means that no domain knowledge is transmitted at this level. The M_2 level is defined by the DSL designer: it is the DSL itself, expressed with M_3. The DSL user models one or many M_1 to represent the M_0 system.

However, this architecture might bring about some confusion because the classical 4-layered architecture organization is not suited to all kind of languages. For this reason, MOF specifications (see section 7.1) point out that MOF and other OMG standards should not have the perceived rigidness of a 4-layered meta-model architecture, since some other standards use a smaller number of

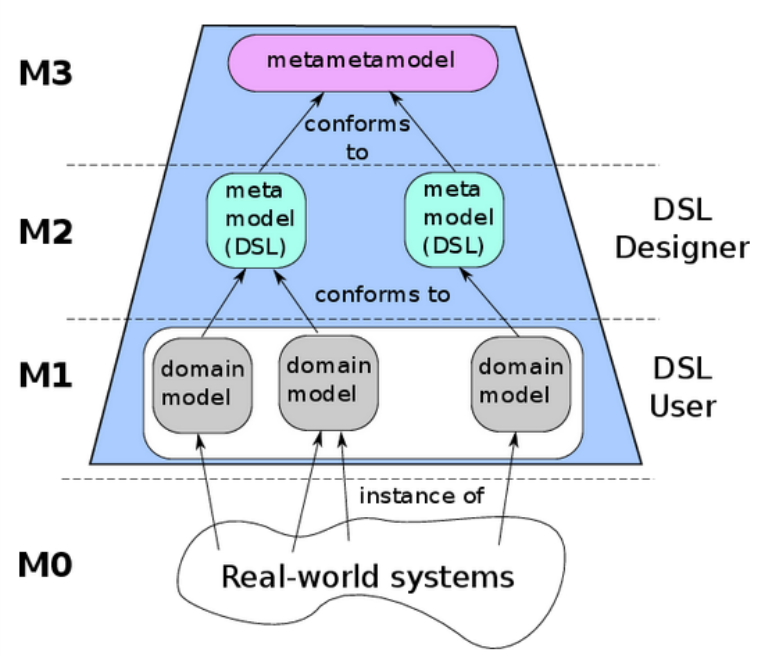


Figure 3.5 – The 4-layered architecture of Meta-Object Facility. Source : European PhD School on Robotic Systems, 2016

layers. Moreover, while there are typically up to four meta-levels, (Mabrouki, 2015) states that some standards may have even more than 4 layers.

3.6 Model Transformation

Another key activity of MDSD is the concept of model transformation. Basically, transformation is the automatic process to transform a source model into a target model through a transformation too. According to (Kleppe et al., 2003), transformation is defined by a set of transformation rules that are "a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language". More precisely, transformation is not restricted to one source and one target, as illustrated in Figure 3.6.

Different kinds of model transformations (Mens and Van Gorp, 2006) are distinguished: endogenous transformation (also known as in-place transformation) modify models conform to the same meta-model and exogenous transformation (also known as model-to-model transformations) translates models between different meta-models. Endogenous transformations are applied for different tasks such as model refactoring, optimization, evolution, and simulation (Kappel et al., 2012), while exogenous transformation are used, for example, for map-ping PIMs to PSMs. Model transformation can also be classified

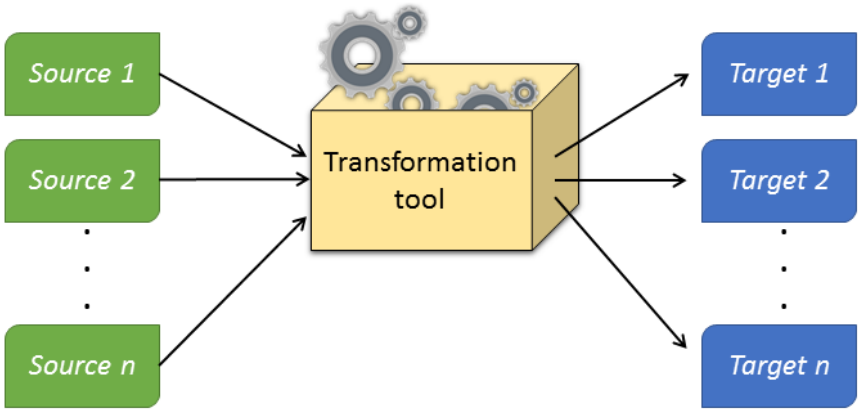


Figure 3.6 – Model Transformation.

by abstraction level. In this case, they can be horizontal (where the source and target mod-els reside at the same abstraction level) or vertical (where the source and target models reside at different abstraction levels).See also (Diaw et al., 2010) for more detail about model transformation taxonomy.

Transformation rules are usually expressed with meta-models, which make rules also applied to each model conform to that meta-model. Ideally, both source and target models should conform to well-defined meta-models. But there is often a need to generate artifacts without an explicit meta-model such as code-generation, or documentation (Touraille et al., 2012). Therefore, MDE tends to make a distinction between two types of transformations: model-to-model transformations (M2M) and model-to-text transformations (M2T). These transformations will be discussed in later sections.

3.7 Model driven engineering

One disadvantage of using general-purpose languages (GPL) is that sometimes, developers are forced to implement non-optimal solutions in order to deliver products that represent more precisely the semantic domain of stakeholders (domain experts). To deal with this issue, MDE is one approach that focuses on tackling the inability of GPL to alleviate this complexity and express domain concepts (Schmidt, 2006). As an integrative approach resulting from years of efforts and experience in software development, MDE combines Domain-Specific Modeling Languages (DSMLs) with transformation engines and generators by delivering the basic principles for the use of models as primary engineering

artifacts throughout the software development life-cycle (Galvão and Goknil, 2007), combined with meta-models and transformations.

DSMLs are modeling languages that incorporate domain expert's concepts into their syntax (abstract and concrete). These DSML provide a higher level of abstraction than a set of tools to produce DSL through models and transformations (also known as artifacts). MDE Transformation engines and generators, in turn, supply the necessary guidelines to execute the appropriate mappings between the produced artifacts. Additionally, model validation, model checking and model-based testing are also part of MDE practices to ensure model quality. Some of the -best known MDE initiatives are Microsoft's DSL Tools and The Eclipse Modeling Project (EMP), the latter being discussed in the following section 3.8

3.8 The Eclipse Modeling Project

3.8.1 Overview

The Eclipse Modeling Project is an initiative that has developed as layers around a central core. Starting from the Eclipse Modeling Framework(EMF) along with the Graphical Modeling Framework(GMF), the project has had an increasing number of contributors in the past decade, with a very active and continuously growing community. The Eclipse Modeling Project focuses on the evolution and promotion of model-based development technologies, providing a unified set of modeling frameworks, tooling, and standards implementations (Eclipse Foundation, 2014). Although the Eclipse Modeling Project hardly mentions MDA at all, it is nonetheless supported to a large degree by the MDA concepts and standards discussed for far.

The Eclipse Modeling project is largely a collection of projects related to modeling and MDSD technologies. This collection was formed to coordinate and focus model-driven software development capabilities within Eclipse (Gronback, 2009). The initial logo of the Eclipse modeling project (figure 3.7) depicts EMF (an older Eclipse project) as the core of Eclipse Modeling Project, enviroined by many components represented by research initiatives, projects, models and capabilities.

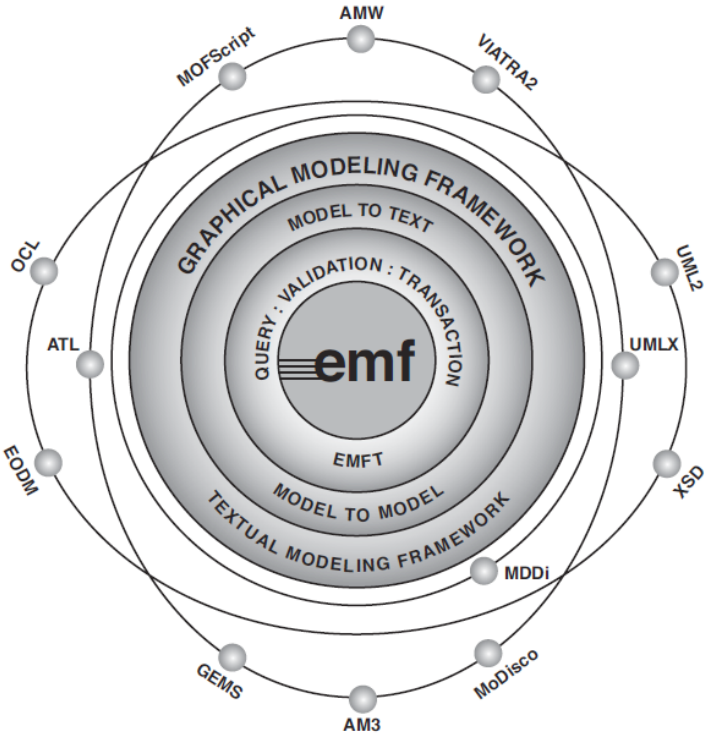


Figure 3.7 – Eclipse Modelig Project and its projects

3.8.2 EMF

The Eclipse Modeling Framework - EMF (Steinberg et al., 2009) provides a modeling and code generation framework for Eclipse applications based on structured data models. EMF is a framework and code generation facility for building Java applications based on simple model definitions. It unifies Java, XML, and UML, where models can be defined using a UML modeling tool or an XML Schema, or even by specifying simple annotations on Java interfaces.

In MOF specifications, 3 compliance points are defined: the complete MOF (CMOF), the Essential MOF (EMOF) and the Semantic MOF (SMOF). The CMOF, as its name states, is the entire specification itself. EMOF is a MOF subset that closely corresponds to the facilities found in object-oriented programing languages and XML. Finally, SMOF (or semantic MOF) was a later request proposal by OMG to add semantic to MOF. More details about these compliance points can be found in MOF's specification (OMG, 2015)

Although EMF supports the key MDA concept of using models as input to development and integration tools, it does not use however any one of the MOF compliance points previously described. Instead, EMF uses ECore, a not fully aligned variant of OMG's EMOF. Essentially, among other elements, an ecore

meta-model allows to define an EClass, an EAttribute, an EReference, and an EDataType. An EClass represents a class, with zero or more attributes and zero or more references. An EAttribute: represents an attribute which has a name and a type. A EReference represents one end of an association (containment or reference) between two classes. Finally, an EDataType represents the type of an attribute (such as an integer, a float or java.util.Date).The Ecore model has a root object representing the whole ecore model. A more detailed version (but not a full one) of the ecore meta-model is presented in Figure 3.8)

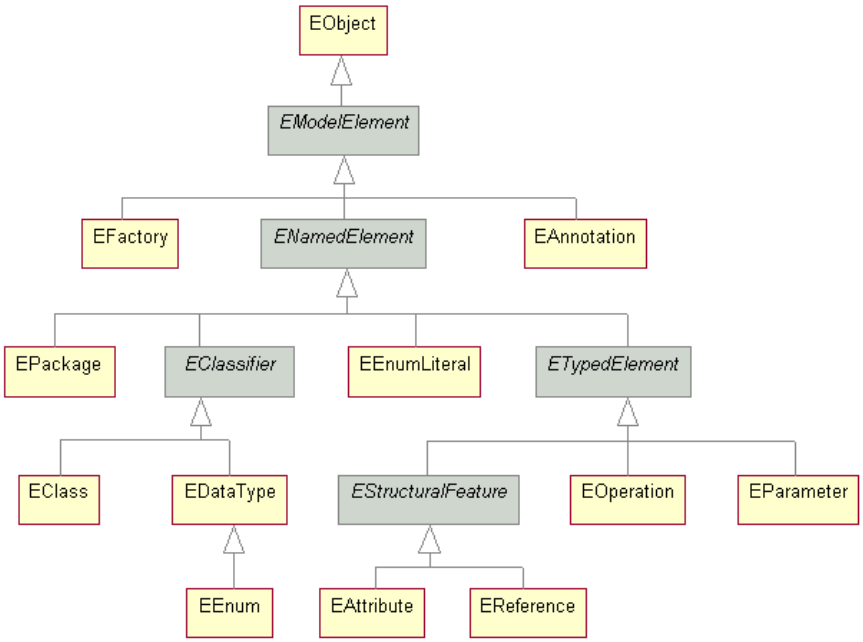


Figure 3.8 – A simplified version of the Ecore meta-model. Source : Eclipse Foundation, 2016(a)

Since EMF's core is the Ecore meta-model, Ecore is used as a meta-meta-model to define DSLs. EMF notably relies on Ecore as a central pivot to enable interoperation between the different tools (Touraille et al., 2012). Because Ecore's aim is to provide mappings from EMOF to Java, the EMF core (Ecore) provides only basic validation and code-generation capabilities (basically only a Java direct translation). To overcome this issue, EMF supplies model transformation and constraints capabilities. Model transformation can be model-to-model (M2M) or model-to text (M2T). Model constraints can be added to a model, and defined either in Java or in Object Constraint Language (OCL), an OMG standard.

3.8.3 Abstract syntax development

One of the key benefits of MDE resides in its meta-modeling capabilities. By using meta-modeling languages, the language designer may define the abstract syntax of his/her own modeling language either a general or domain specific languages (Brambilla et al., 2012). The abstract syntax, made in Ecore, is represented by a meta-model and its development process can be, in its simplest form, a three-step iterative and incremental process (Brambilla et al., 2012) as illustrated in Figure 3.9

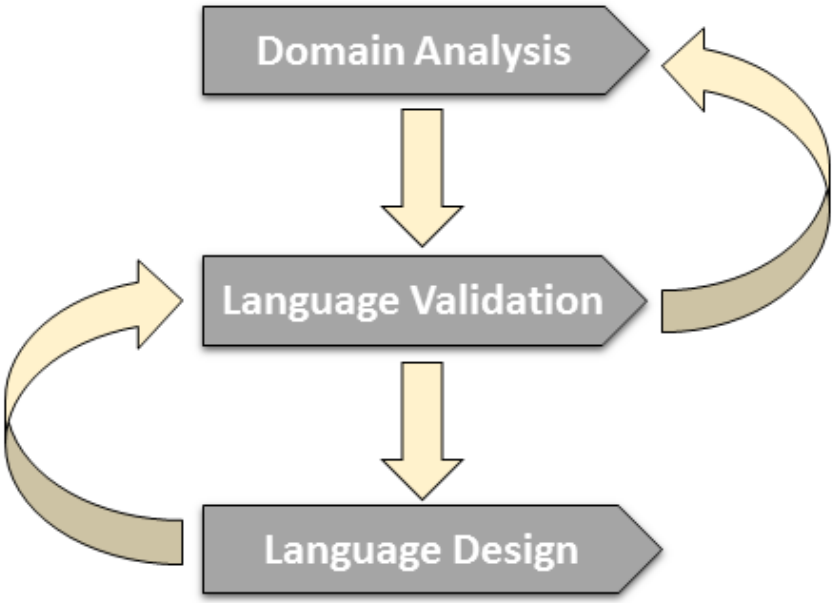


Figure 3.9 – The meta-modeling process. Source : Eclipse Foundation, 2016(a)

- Modeling domain analysis: technical literature, existing implementations, customer surveys and expert advice are some of the required elements for domain analysis, according to (Prieto-Diaz, 1990). Modeling domain analysis can be seen as a process where information used in developing software systems is identified, captured, structured, and organized for further reuse. Reuse is performed through the use of infrastructure, such as domain models, development standards, and repositories (libraries) of reusable components.
- Modeling language design: this is the process that formalizes the concepts in the abstract syntax into a concrete syntax. Later, some constraints are

specified in OCL in order to remove language ambiguities and add well-formedness rules. In that case, some feedback of domain experts may be needed, wherever the concrete syntax should be an accessible modeling language.

- Modeling language validation: this is an essential process in MDE to check whether a model meets the informal requirements a developer has in mind. It pro-vides the necessary feedback to the next iteration step in meta-model development process.

In MDE, the abstract syntax is used in the development of almost every artifact that follows, including text and graphical concrete syntax, model-to model transformations, and model-to-text transformations (Gronback, 2009). Since EMF relies on Ecore, the abstract syntax is defined in an Ecore meta-model. There-fore, the creation and edition of an Ecore meta-model can be performed in several ways: by using a tree-like editor to add, remove or move model elements and modify their properties; by importing a set of annotated Java class, by importing a xml file; by importing UML models (Rational Rose. mdl file format is recommended, but other formats are also usable) or still; by using EcoreTools [2], a graphical tool quite similar to an UML editor.

By default, EMF uses XMI (XML Metadata Interchange), an OMG standard for-mat for exchanging metadata information via Extensible Markup Language (XML). Additionally, since Ecore is a representation of the persistent XMI format, textual editing is also possible through the use of OCLinEcore[3], an Ecore editor with highlight syntax for editing OCL constraints.

3.8.4 Concrete syntax development

Computer languages are always processed by a compiler. One of the main functions of a compiler is to process statements written in a particular computer language and turn them into machine language. This process is usually divided into lexing and parsing. In lexing, the compiler performs the lexical analysis to generates tokens from each word contained in a string. In the parsing process, (also called syntax analysis), the compiler checks whether a sentence is grammatically correct according to a formal grammar by identifying the function of each generated token. The whole process is called the parsing process (figure

[2] http://www.eclipse.org/ecoretools/index.html
[3] https://wiki.eclipse.org/OCL/OCLinEcore

3.10) aand is executed by a parser. Although parsers can be manually designed, using a parser generator affords some advantages, such as a higher level of program-ming abstraction and the capacities to generate the target parser in multiple languages (Ghosh, 2010). Many parser generators are available today (such as YACC, BISON, ANTLR, etc.). Usually, the function of a parser generator is to produce parser trees. Parser trees represent the syntactic structure of a string according to some grammar and some custom actions that are executed in recognition of those rules. Differently from abstract syntax trees (AST), which are an abstract representation of the input grammar, parser trees retain all of the information on the concrete structure of the input grammar.

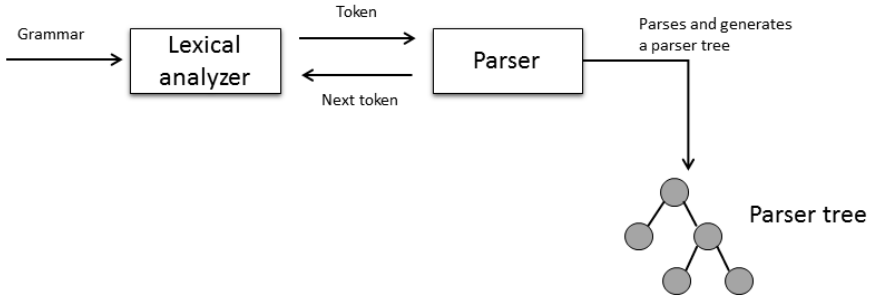


Figure 3.10 – The parsing process

While the abstract syntax includes those concepts that are represented in the language and the relationships between those concepts, concrete syntax definition provides a mapping between meta-elements and their representations for models (Challenger et al., 2014). AAbstract syntax only consists of the structure of data, while concrete syntax also includes information about the representation (Figure 3.11). In other words, the abstract syntax represents which concepts are present in a language and how they relate to each other.

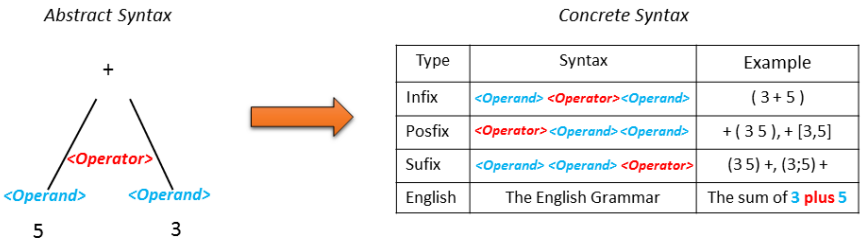


Figure 3.11 – Abstract and concrete syntax

The concrete syntax, on the other hand, is defined by a grammar, which is a set of rules that specifies the syntactic structure of the DSL. These rules are made from the terminals (symbols of the alphabet taken into consideration) and

non-terminals represented by variables ranging over strings of terminals. In Chomsky hierarchy[4], , context-free grammars are type-2 grammar rules that are produced by $A \rightarrow \beta$, where A is single non-terminal symbol and β is a string of symbols. For a more detailed view of the different grammars contained in Chomsky hierarchy, see section 2 in (Jäger and Rogers, 2012).

Nevertheless, in order to express context-free grammars, some meta-syntax notations are used to obtain a formal description of a formal language. One example of a meta-syntax notation is Backus–Naur Form (BNF). It was initially developed by John Backus in an effort to develop the FORTRAN language. Later, Peter Naur, an editor of the ALGOL report, popularized the BNF notation, using it to describe the complete notation and describe the complete syntax of ALGOL (Pattis, 2013). With some modifications, the Extended form of the Backus–Naur Form (called EBNF) has today many variants of the original ISO EBNF (Information technology Syntactic metalanguage, 1996) and is largely employed as formal notation to define programming languages.

In MDA, MOF plays exactly the same role as EBNF (as a meta-syntax notation) to define programming language grammars. MOF is used to define metamodels, just as EBNF is a DSL to define grammars. Similarly, to EBNF (that can be used to define itself), MOF could be defined in MOF. Because of similarities between MOF layers and concrete syntax development using BNFs notations, concrete syntax definitions can be easy mapped to EMF environment. Also, EMF provides a concrete syntax which can be textual or graphical concrete syntax, respectively supported by the TMF and GMF projects.

3.8.4.1 TMF

The Textual modeling framework (TMF) is an EMP's project aiming to support the development of textual concrete syntax. TMF is based on a meta-model and syntax specification, offering several functionalities that include a parser that reads the textual representation of the model and instantiates the corresponding EMF model, an eclipse text editor that supports syntax highlighting, code completion, navigation, and other features. Moreover, both the editor and the parser are able to plug in various constraint engines (such as OCL) in order to provide model validation.

[4]The Chomsky hierarchy is a collection of four classes of formal languages, each of which is a proper subset of the classes above it

These and many other objectives were part of the Textual Concrete Syntax - TCS (Jouault et al., 2006) available in the early years of TMF project. Although currently inactive, much of the TCS project's philosophy has simultaneously evolved into another TMF project termed Xtext, in an attempt to speed up grammar development and ease synchronization between the meta-model and grammar. Xtext is a framework for programming languages development through a grammar language to implement textual DSL. Grammar language provided by Xtext is similar to EBNF, but with additional features (such as cross-references) to achieve similar expressivity as an ecore-based meta-modeling language.

In fact, Xtext uses a specific implementation of the ANTRL[5]for parsing input files. As an output, Xtext framework generates a meta-model (by default), an editor, and an abstract syntax tree (AST). Moreover, because Xtext heavily relies on EMF, the generated AST is implemented in EMF and can be re-used and integrated with other EMF-based editors such as GMF. This mechanism is illustrated in Figure 3.12.

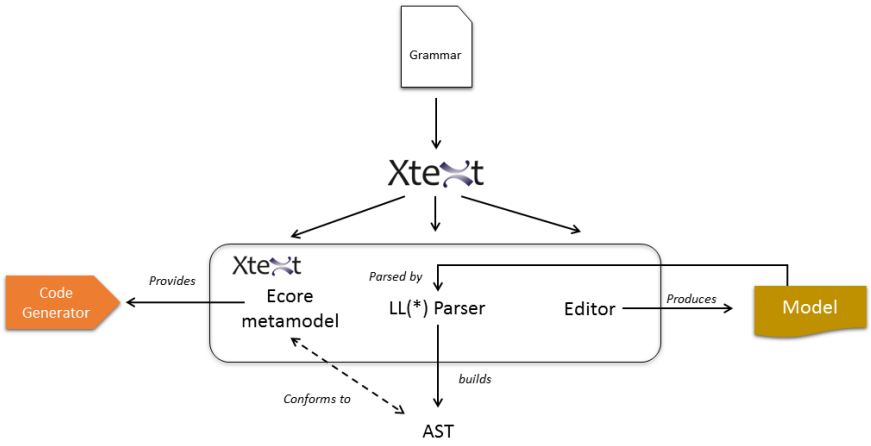


Figure 3.12 – Abstract and concrete syntax. Adapted from : Jan Köhnlein, 2009

Although not officially part of TMF, other DSL textual modeling frameworks are also available for EMF environment, MontiCore[6], Eclipse IMP (currently marked as an archived eclipse project), and Textual Editing Framework (TEF). All these frameworks use a text-parser-model paradigm and basically implement the background parsing strategy (like Xtext). TEF, on the other hand, used a slightly different approach, where a kind of graphical editor is provided and using Model-View-Controller (MVC) technique, the textual model representation is separated into smaller sub-views for each model element. This MVC

[5]Differently from YaCC, that is a LARL parser generator, ANTRL is a LL(*) parser generator
[6]http://www.monticore.de/

pattern is implemented throught eclipse's Graphical Editor Framework (GEF), allowing graphical and textual notations to be combined.

3.8.4.2 GMF

The GMF [7] project is a set of tools that combines both EMF and GEF[8] for building graphical concrete syntax with eclipse-based functionality. GMF developers have mostly adopted the term 'toolsmith' to refer to developers that use GMF to build plug-ins. One of the reasons is due to the straightforward process to use tools to create graphical concrete syntax: the toolsmith first defines the graphical definition (i.e. visual aspects of the generated editor), the tooling definition (editor palettes, menus, etc.) and finally, the mapping definition between the meta-model and visual model (graphical and tooling definition).

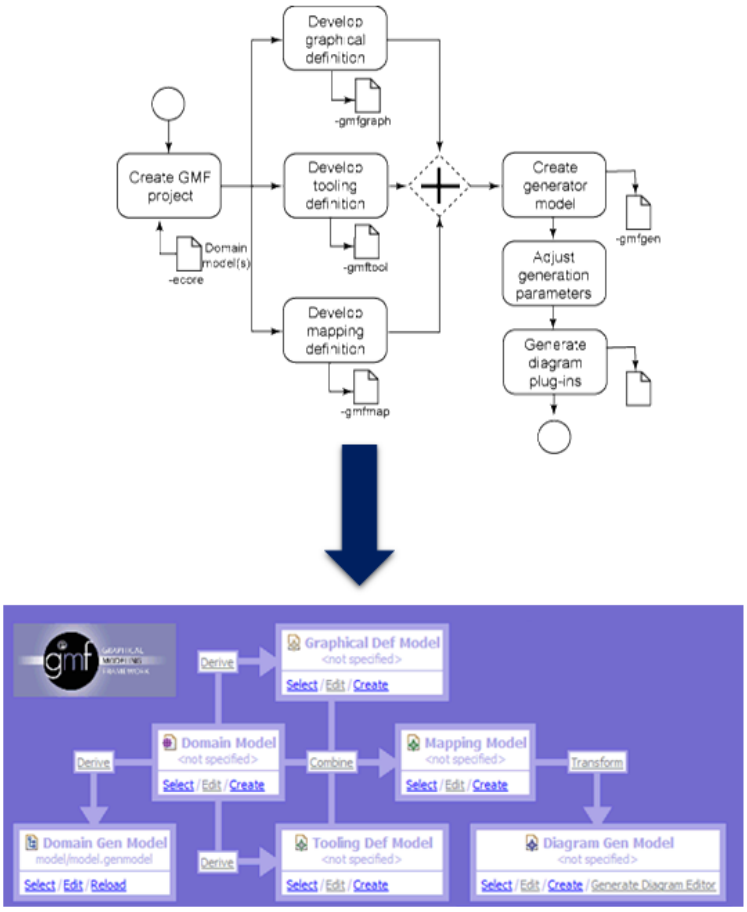


Figure 3.13 – The GMF process for generating graphical concrete syntax. Adapted from Eclipse Foundation, 2016(b)

[7] Graphical Modeling Framework, also called Graphical Modeling Project (GMP) : http://www.eclipse.org/modeling/gmp/

[8] Graphical Editing Framework - a framework to create rich graphical applications in the Eclipse Workbench : https://eclipse.org/gef/

Although a quick editor could be rapidly produced from this process, any future modifications should then be done either by hand (requiring a good knowledge of GEF objects) or by repeating the whole process over and over until the editor is conform to the toolsmith's specification.

To improve this method, EuGENia[9] is a tool that automatically generates .gmfgraph, .gmftool and .gmfmap models needed to implement a GMF editor. Eugenia uses the Emfatic[10] language to annotate Ecore meta-models and include elements to generate the graphical editor. With that approach, much of the graphical editor customization can be easily programmed by annotating a single Ecore meta-model with EuGENia. However, one drawback of EuGENia's approach is the available number of notations: since EuGENia does not support all GMF features (as it should not, lest it would be as complex as GMF), modifying a graphical editor feature that was not supported by the notations provided by EuGENia is not a simple process.

In parallel, focusing on decreasing the complexity of specifying graphical editors based on the GMF, some other projects arose as an alternative to the pure GMF or Eugenia approaches. One of these projects is the Papyrus[11] modeling environment, a graphical editing tool for UML2 targeted at implementing the whole OMG specification. Although the initial objective of the Papyrus project was to provide a UML editor in the context of EMF, Papyrus also offers a customizable environment to define a toolsmiths' own graphical, textual or tabular notation, to change existing palettes and many other features.

Also, intenting on facilitating the building of graphical concrete syntax, Sirius[12] provides a graphical designer editor where the toolsmith can specify a variety of and customizable viewpoints adapted to the user's role or activity. Specifying different viewpoints in Sirius allows, for instance, a employee of a department to visualize a graphical concrete syntax (i.e. table format) in a different way than his superior (i.e. icon-trees). All those points-of-view can be specified from the same ecore meta-model. Metamodel elements can be graphically selected or parsed using the Acceleo Query Language (ACL). Additionally, Sirius allows "services" to be specified in Java, enabling to easily combine do-main elements with additional data that are sometimes retrieved from outside of the domain model (like date from external databases).

[9] http://www.eclipse.org/epsilon/doc/eugenia/
[10] A language designed to represent EMF Ecore models in a textual form
[11] https://eclipse.org/papyrus/
[12] https://eclipse.org/sirius/

3.8.5 Model-to-text transformation

One of the aims of M2T transformation languages is to improve code-generation by tackling some drawbacks of GPL code-generators, such as the lack of declarative query languages and reusable base functionality. These transformation languages use a template-based approach (not to be confused with template programming). Templates are composed of meta-markers (for the dynamic part), where they (the templates) query additional data sources (in this case, user models) to produce text through a template engine. The process is summarized in Figure 3.14

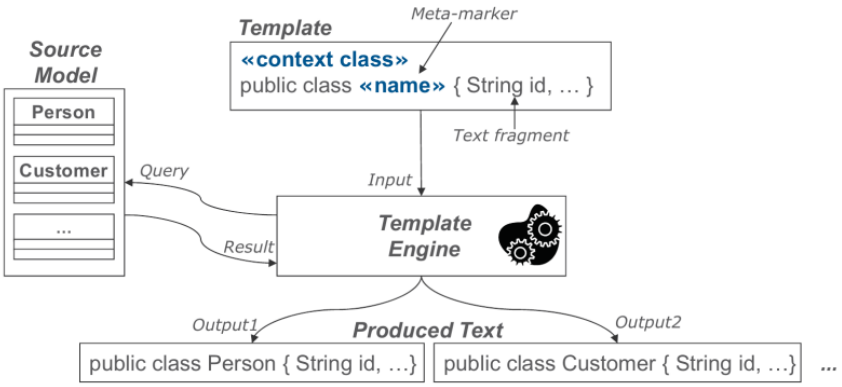


Figure 3.14 – Template approach mechanism in M2T. Source : Brambilla et al., 2012

In M2T projects, the most common available frameworks to perform model transformation into text are JET, Xpand, and Acceleo. JET is a code generation framework that offers facilities that are used by EMF. In Jet, JSP-like template files can be edited and transformed into any kind of source artifact. Xpand is a template engine, similar to FreeMarker, Velocity, JET and JSP. However, it features some very unique properties that make using Xpand very suitable for generating code from models, such as type safety and polymorphic dispatch. Acceleo, on the other hand, is a pragmatic implementation of the MOF Model to Text Language standard[13], an OMG's standard that addresses how to translate a model to various text artifacts such as code, deployment, specifications, etc. With code highlighting and auto-completion, Acceleo syntax is also based on OCL's syntax standard.

[13]MOFM2T - http://www.omg.org/spec/MOFM2T/1.0/

3.8.6 Model-to-model transformation

TThe Model-to model transformation (also known as M2M or MMT) is a project that hosts Model-to-Model Transformation languages. With these M2M languages, we can define model transformations to produce other models or generate textual output. In general, M2M language mechanism (Figure 3.15) consists in taking a model (that conforms to a source meta-model) as an input, specify some transformation rules using an M2M language, and generate a target model (that conforms to a target meta-model).

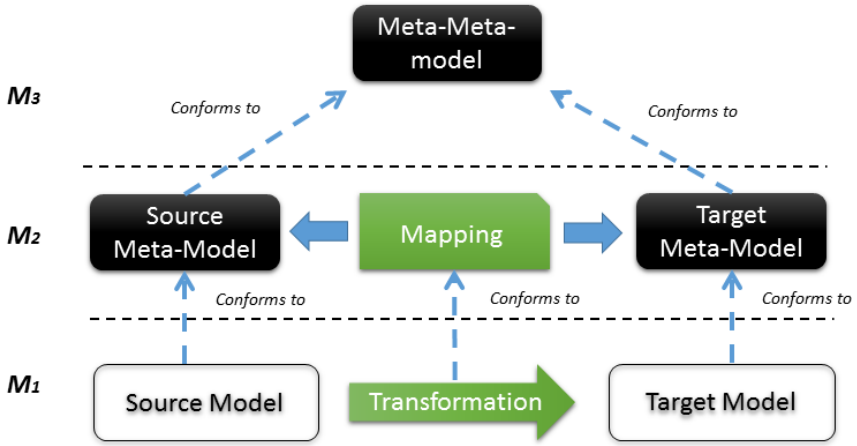


Figure 3.15 – Model to model transformation (M2M)

BBut in some specific cases, some M2M techniques can be considered. Some of these techniques include: Model Refactoring (when the source and target model(s) belong to the same level of abstraction and are also conform to the same meta-model), Model Migration (when source model needs to be updated to a target model to re-establish conformance in response to meta-model evolution), and Model Merge (when a set of source models must be transformed into a target model that conform to a meta-model resulting from merging all source meta-models). More details about the implementation of these techniques can be found in (Gronback, 2009).

In order to perform M2M transformation, MDE provides M2M transformation languages to implement mappings between models. Some of these languages include QVT (Query/View/Transformation), ATL Kermeta, and Xtend. Their general characteristics are presented in Table 3.1.

The **Q**uery-**V**iew-**T**ransformation standard of OMG covers three languages for developing model transformations : QVT-**R** (for Relations), QVT-**C** (for Core) and QVT-**O** (for Operational). The former two are declarative (a description

Table 3.1 – Model-to-model technologies available in EMF

MM2 Language	Maintained by	Features
ATL	OBEO and INRIA	Deals with heterogeneous data. Provides imperative constructs with virtual machine for executing model transformations
QVT	OMG	Provides a declarative language to specify mappings on MOF models (xmi to xmi)
Xtend	Eclipe Foundation	Statically-typed programming language with its roots in the Java programming language. Can also be used as code generation language
Kermeta	INRIA, CNRS, INSA	MOF compliant, model oriented, imperative, statically typed and object-oriented language

of what to investigate, the implementation being left to an interpreter), while the latter is imperative and a direct implementation of transformation instructions. QVT-**R** is a declarative approach for both unidirectional and bidirectional model transformations, while and QVT-**C**, a small language designed to define semantics in a more simple way. Finally, QVT-**O** is a unidirectional imperative language: it is designed for writing unidirectional, operational mappings where there is a source model and one or more target models. Operational mappings are either transformed to the Relations language and then to the Core language, or directly to the Core language (Barendrecht, 2010)

An editor and parser for both QVT-**R** and QVT-**C** is available in the QVT Declarative for Eclipse[14] while QVT-Operational has also his own editor[15]. QQVT however does not cover what is not considered by OMG as a model transformation, view or query (e.g transformations to or from textual models). Be-cause ATL deals with heterogeneous data, it provides some solutions to QVT transformational problems. Moreover, ATL offers imperative constructs to facilitate the specification of mappings that can hardly be expressed declaratively.

Xtend2 (or Xtend for simplicity), is considered as a successor to Xtend which allows having Xpand's template syntax as an expression. Xtend is actively used

[14] http://wiki.eclipse.org/MMT/QVT_Declarative_(QVTd)
[15] https://wiki.eclipse.org/QVTo

by Xtext to implement custom scoping and validations rules. But since version 2, Xtend's focus has changed from code generation to a java-like GPL. Consequently, many features that are common in template-based language environments are not provided, such as component generators to observe how the generated code looks like and where to produce output codes when templates are invoked. Nevertheless, Xtend is still used by many developers as a code generator language.

Lastly, Kermeta is an imperative language for modeling, with a basic syntax inspired from Eiffel. Its compliance with MOF is based on EMOF level, and it is based on Xtend since version 3. Additionally, some other M2M languages include: MoTE, a model transformation engine developed by MDELab[16] to model Triple Graph Grammars (Schürr, 1994) and perform model transformations and synchronizations; the **J**anus **T**ransformation **L**anguage, developed by the University of L'Aquila[17]; the **E**psilon **T**ransformation **L**anguage[18], maintained by the Epsilon project; and the Ruby **T**ranformation **L**anguage, an extension of the Ruby programming language[19] for model to model transformation.

3.9 Conclusion

As a vast set of tools and approaches developed over the past decades, Model Driven Engineering is a software development methodology designed through many years of combined experience from both industry and academia in software engineering. Born as an effort to define standards and tools to facilitate and cover all stages of software development, MDE is currently one of the main approaches in software engineering to deliver quality and reliable solutions, having applications in a plethora of domains. Moreover, with the advances in MDE, the process of developing DSLs has become more transparent. Combined with MDE's "everything is a model" philosophy, products are more easily developed, granting faster feedback from domain-experts.

[16] https://www.hpi.uni-potsdam.de/giese/public/mdelab
[17] http://jtl.di.univaq.it/index.php
[18] http://www.eclipse.org/epsilon/doc/etl/
[19] http://rubytl.rubyforge.org/

Chapter 4

Modeling social-ecological systems

4.1 Introduction

Social-ecological systems(SES) can basically be understood as the link between social and ecological systems. Essentially, SES researchers seek to understand the relations between institutions, actors, types of environmental ecosystems and ecosystems services. Grasping the relation between these elements naturally leads to an engagement of many different disciplines in SES study. These disciplines are in a constant effort to share a common vocabulary for the construction and test of alternative theories and models (McGinnis and Ostrom, 2014), that in turn, may become more and more complex.

In this chapter we introduce the specification of a MAS model called ECEC, short for **E**volution of **C**ooperation in an **E**nvironmental **C**ontext. Previsouly described by (Pepper and Smuts, 2000) our specification is a simplified replication of the authors' model. The ECEC model was chosen for two reasons. First, the model specification is platform-independent: the model is elucidated by using a natural language (i.e. English) where the model's description is realized without mention of any programing language or MAS platform. Second, the model possesses different types of behaviors, which in turn, are described in a different way. And it is precisely these different manners of describing a model that is very often found in SES modeling. The model discussion will serve as a through-line to Chapter 5.

In this short chapter, we will describe in detail the ECEC model and the agents involved in the model. Next, we shall focus on their behaviors, and how their initial values are set and described

4.2 The ECEC Model

In the domain of evolutionary biology, there is a high interest in understanding how fostering selfish (individualistic) and altruistic (or cooperative) behavior can influence natural selection and benefit group members under particular demographic conditions. These members can organize themselves into "small trait groups", a collection of individuals that influence one another's behaviors. In nature, traits usually aim at reproductive benefits, increased chances of feeding and higher protection from predation. Those traits commonly result in cooperative behaviors, with many examples found in nature, such as schools of fish, flocks of birds or herds of bisons.

The ECEC model thus aims at investigating the evolution of two different cooperative traits: alarm calling and feeding restraint. Alarm calling is a trait which benefits only individuals near the alarm-caller (i.e. prey presence, natural resources proximity, etc.). This trait is uniquely taken for self-benefice. Feeding restraint, on the other hand, considers that an individual action takes into consideration the common interest of the group to which an individual belongs. This is the classic example of altruistic behavior among non-humans. or prudent predation behavior found in certain type of animals.

The majority of works that try to explain those behaviors through models, are mostly quantitative models of group selection or, alternatively, make use of systems of equations (Pepper and Smuts, 2000). However, there are some limitations in modeling traits based on purely mathematical models[1]. Instead of using equations, the authors advocate the use of a multi-agent systems approach to pursue their investigation, due to MAS ability to represent the behaviors and interactions of individuals in a more direct and natural way, to incorporate variation over space and time, and to incorporate non-linear dynamics capacity.

To represent the ECEC model using MAS approach, (Pepper and Smuts, 2000) considered a model consisting of a two dimensional grid, wrapped in both axes (to avoid edge effects) containing two kinds of entities: plants and foragers. The main idea is the study of the survival of two populations of agents that depends on the spatial configuration. The representation of ECEC is depicted in Figure 4.1

[1] According to (Williams, 1966), those type of models either require simplifying assumptions, such as homogeneous randomly mixed populations and infinite population sizes, or in some cases, population structure (the division of a population into more or less discrete groups) must be assumed a priori.

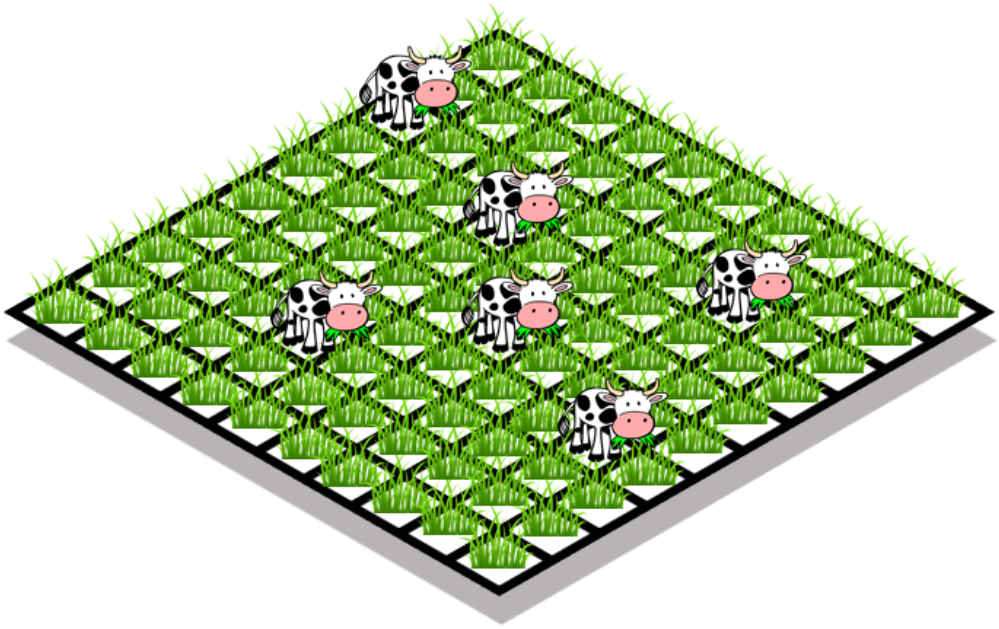


Figure 4.1 – ECEC representation : Foragers distributed in a spatial grid of plants

4.2.1 The plant and its behavior

The Plants are created only once and have a fixed location. They do not move, die, or reproduce. A plant's only behavior is to grow (and be eaten by foragers). The plants vary only in their biomass, which represents the amount of food energy available to foragers. At each time unit, this biomass level increases according to a logistic growth curve:

$$X_{t+1} = X_t + rX_t\left(1 - \frac{X_t}{K}\right)$$

where X_{t+1} is the plant's biomass over time, r is reproduction rate, and K the capacity rate.

4.2.2 The Foragers and its behavior

In order to study two types of behaviors with different traits, the model considered two types of Foragers: Restrained foragers and Unrestrained foragers. Their only difference is their feeding behavior, explained in section 4.2.4. The foragers, thus, have the following common behaviors: they consume energy, they move, they feed, they reproduce and, eventually, they die.

4.2.3 The foragers' energy consumption behavior

At each step, the Foragers burn energy according to their catabolic rate. This rate is the same for all foragers. It is fixed to 2 units of energy per time period. Foragers lose energy (catabolic rate, 2 points) regardless of whether or not they move.

4.2.4 The foragers' feeding behavior

When "Restrained" foragers eat, they take only 50% of the plant's energy (biomass). In contrast, when "Unrestrained" foragers feed, they take 99% of the plant's biomass, so that plants can continue to grow after being fed on, rather than being permanently destroyed. However, when all foragers eat, they feed on the plant in its current location, increasing their own energy level by reducing the same amount of the plant's biomass.

4.2.5 The foragers' reproductive behavior

Foragers reproduce if their energy reaches the fertility threshold (100 energy units). Their offspring keep the same heritable traits (i.e. same feeding behavior). In that case, it reproduces asexually, creating an offspring with the same heritable traits as itself (e.g. feeding strategy). At the same time the parent's energy level is reduced by 50 energy units, the offspring's initial energy (50 energy units). The newborn offspring will occupy the free place nearest to their parent.

4.2.6 The foragers' move behavior

In their search for food, Foragers examine their current location and their surroundings. From those locations not occupied by another forager, they choose the one containing the plant with the highest biomass. If the chosen plant yields enough food to meet their catabolic rate (2 units of energy) they move there. If not, they move instead to a randomly chosen adjacent free place not occupied by any forager. This movement rule leads to the migration of foragers from depleted patches, and simulate the behavior of individuals exploiting local food sources while they last, but migrating rather than starving in an inadequate food patch.

4.2.7 The foragers' die behavior

If its energy level drops to zero, a forager dies. Foragers do not have maximum life spans.

4.2.8 Model initial values and execution

The model is executed in the following order: the plants grow, then, foragers eat, then reproduce, then move, then die (if the energy level reaches zero). That sequence of behaviors can also be visualized in the form of an activity diagram, as shown in Figure 4.2

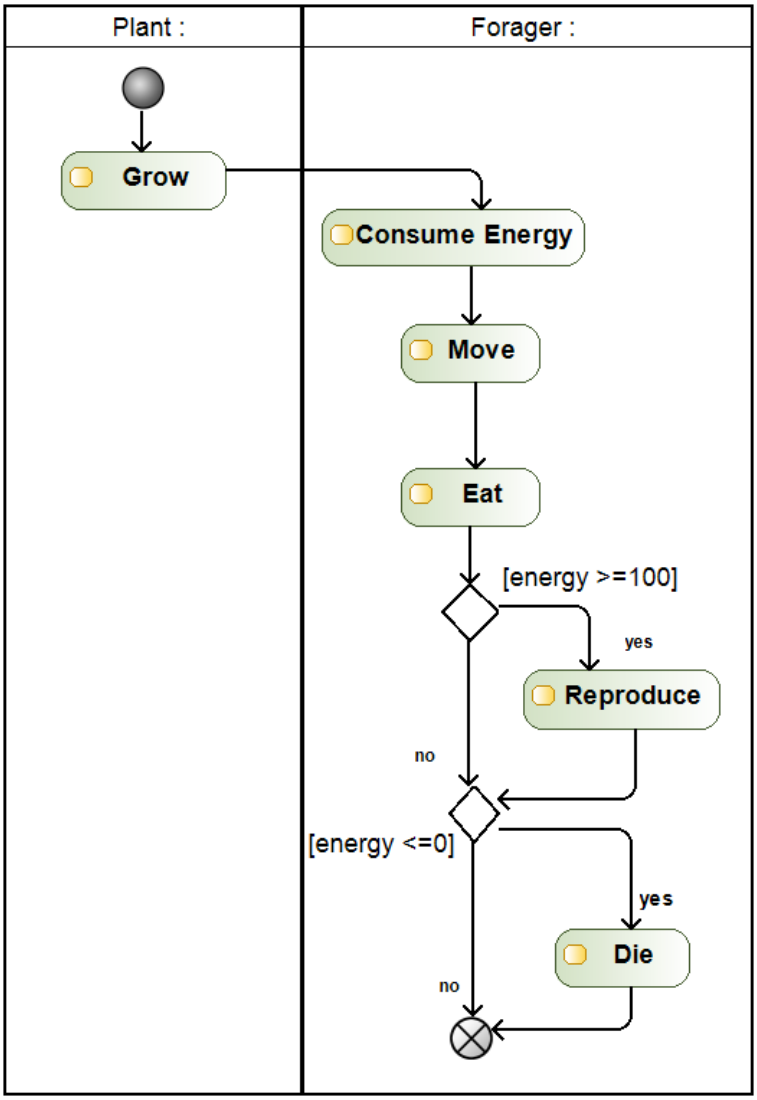


Figure 4.2 – Activity diagram representing the sequence of behaviors to be executed during an ECEC model simulation

Table 4.1 – ECEC model's initial values

Variable	Entity	Initial value
Biomass	Plant	Random value between 0 and K
K	Plant	10
r	Plant	2
Catabolic Rate	All foragers	2 units of energy
Fertility Threshold	All foragers	100 units of energy
Energy	All foragers	50 units of energy
Harvest rate	Restrained Forager	0.5 (or 50% of the plant's biomass)
Harvest rate	Unrestrained Forager	0.9

In order to initialize the model, some initial values are required. The plant must have initial values for its biomass, K and r. Those variables are used in the growth behavior of the plant. Foragers also possesses initial values that are used in the description of some of their behaviors. The catabolic rate is the amount of energy that foragers lose over time and has a default value of 2. The fertility threshold is the max energy value of a forager before it reproduces. It is fixed as 100. Finally, the initial energy of all foragers is set as 50. Those values are listed in Table 4.1

4.3 ECEC behavior representation

In ECEC, behaviors may be expressed through 3 different ways: through equations (like in the "grow" behavior of the plant), through a list of activities (like in the main behavior of ECEC or other behaviors) and as activity diagrams (like the "eat" behavior of a foragers). Each behavior is characterized by using a different set of elements to describe itself. For instance, equations use numbers and letters combined with arithmetic operations. Activity diagrams use graphical UML symbols to represent the flow of a behavior. A behavior declared as a list of activities employs the natural English language to describe a set of actions. The difference between these behaviors is depicted in Figure 4.3, and that

figure will be used in chapter 5 to explain in more details the elements that are part of each of these behavior representations.

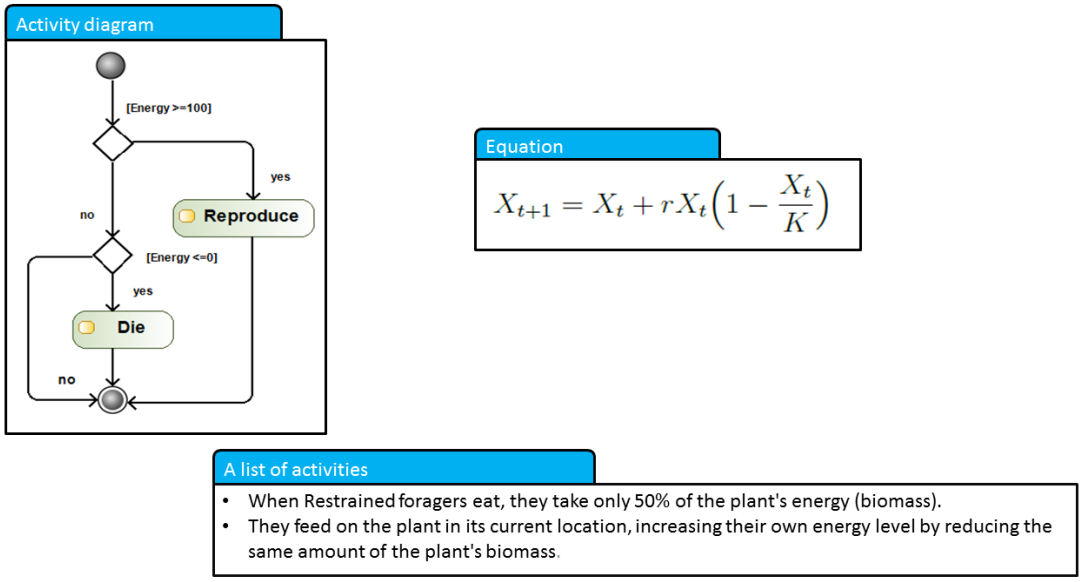


Figure 4.3 – Diffrent ways to represent a behavior in ECEC

4.4 Conclusion

In this chapter, we introduced a model called ECEC, a model in the domain of biology that aims to investigate the survivability of two populations with different traits feeding on a common resource. To illustrate the idea of the model, ECEC was conceptualized based on MAS approach, where individuals influence other individuals through their relations with an environment. Although it is a limited and simplified version of the original model, the model description is completely independent of a programming language, and does not require any previous understanding of any MAS platform. Consequently, the model explained in this chapter will be used to illustrate how we implemented the language that we propose in the following chapter, which is the main contribution of this work. Through the ECEC model, we discovered that some of the elements contain discourses used by domain experts to model reactive behaviors on SES. The concepts presented in this chapter will be used to demonstrate the conception of a DSL that we propose in the next chapter.

Chapter 5

B-Reactive - A DSL to model reactive behaviors in MAS

5.1 Introduction

In this chapter, we propose a DSL to model reactive behaviors for MAS, called B-reactive. During the conception of B-reactive, our objective was to build a DSL focusing SES models. As seen in Chapter 3, developing a domain-specific modeling language essentially involves three main steps: the definition of an abstract syntax, the specification of a semantic domain and the design of a concrete syntax. We first define our semantic domain in section 5.2. To define the semantic domain of our language, we used the ECEC model described in Chapter 4 as a guide to illustrate some of the common terms found in SES, as well as how behaviors are defined in those type of models. Next, we use UML to define B-reactive's abstract syntax, based on the semantic domain discussed in section 5.2. Finally, in section 5.4, we propose a concrete syntax that is explained through a set of different examples. With that, we provide the basic structure of a concrete syntax that can be used to model reactive behaviors of MAS.

5.2 The Semantic domain

The semantic domain is related to the domain on which the description of a language is based. The semantic domain is a layer that provides a clear separation of concerns between parsing a language and the resulting semantics. It is usually defined in two different ways: through informal semantics (by using plain English) or through formal semantics (by using some mathematical notation

formalism such as operational semantics, axiomatic semantics or denotational semantics).

For some, it is often argued that an accepted mathematical formalism (such as CSP or Z) should be the starting point for defining a computer language (e.g. LISP), since informal semantics although easier to read, can easily be misinterpreted. However, when it comes to modeling languages such as UML, applying a mathematical formalism to describe semantics might not be the most suitable approach (Selic, 2004). One of the reasons might be the fact that there is still no consensus about the UML semantic definition. In particular, some fundamental premises regarding the nature of UML behavioral semantics are missing: it only deals with event-driven or discrete behaviors. This means that continuous behaviors, such as those found in many physical systems, are not supported.

Nonetheless, UML is one of the keystones of MDA and for that reason, the abstract syntax defined in section 5.3 was conceptualized making use of UML. Moreover, behaviors were defined through an abstract representation of the UML activity diagram. Therefore, diving into a formal semantic definition of many UML aspects would be out of the scope of this work.

Before we provide an informal semantic definition of our proposed language, we first analyzed some terms contained in SES domain. Using the ECEC model description as an illustration, a model can be defined according to the general structure depicted in Figure 5.1

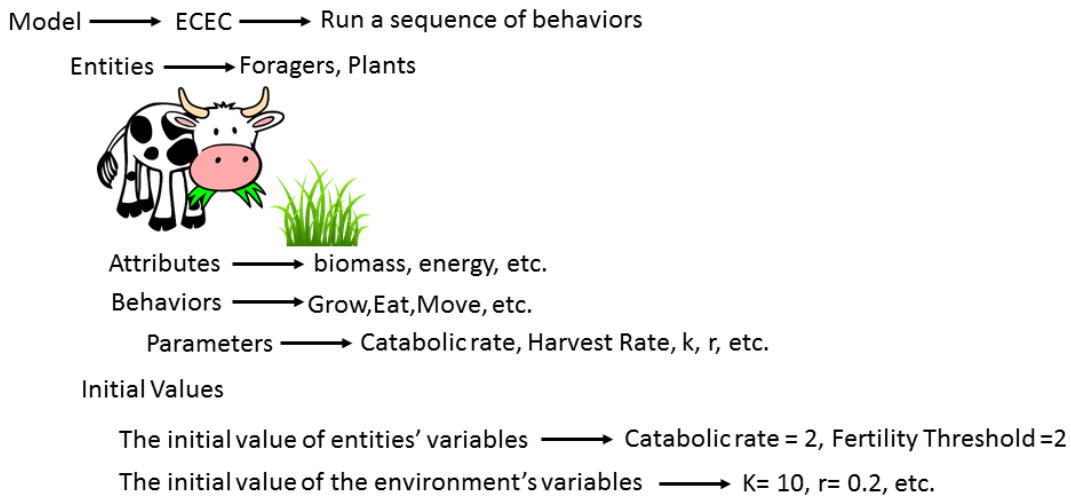


Figure 5.1 – A model definition and its elements

In Figure 5.1, models usually have a name (i.e. ECEC) and use a sequence of behaviors (i.e. grow, then eat, then move, etc) to define in which order the

model will perform activities. This model is also composed of entities (i.e Plant, Forager), that in turn, have attributes (i.e. energy, biomass) and behaviors (such as “grow”, “eat”, etc.). The model also defines initial values (k=10, r=0.2, etc.) for their entities. These entities may represent the environment (like the Plant) or Agents (like Foragers). Nonetheless, this general structure only defines what the model contains and how behaviors are defined.

By deepening the analysis of how behaviors are explained, let us consider the foragers’ behaviors of eating and energy consumption. In the first behavior (eating), foragers take up only 50% (if restrained) of the plant’s energy (biomass). By feeding from the plant, they increase their own energy level by reducing the same amount of the plant’s biomass. In the second, foragers loose 2 energy units at each time step. Intuitively, these behaviors could be replaced by the simple action of Adding something to an attribute (in the case of taking the energy from the plant) or Removing something from an attribute (in the case of an grazed plant loosing its biomass or the forager loosing its energy over time). The idea of using these terms to express certain types of behaviors are illustrated in Figure 5.2

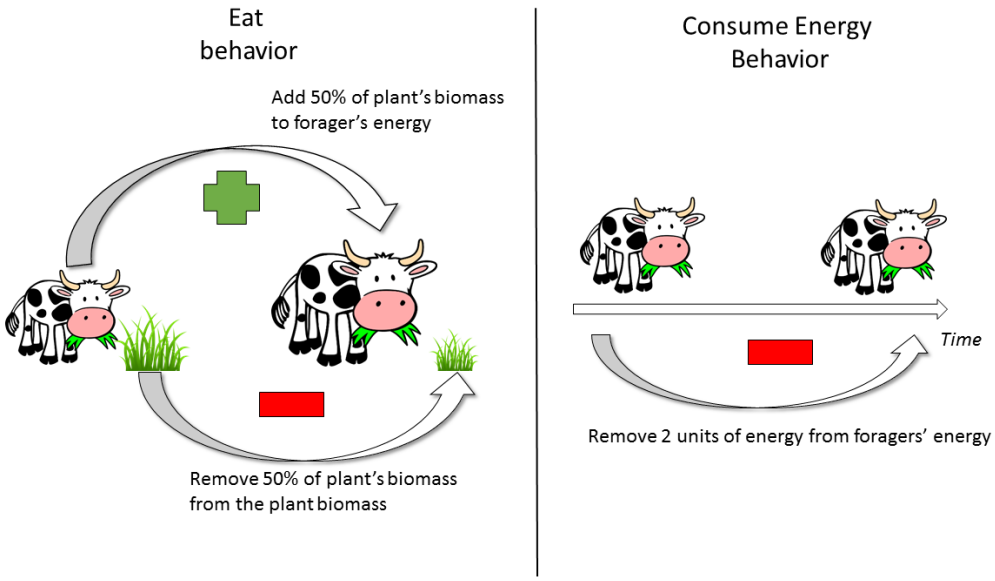


Figure 5.2 – A different way to explain behaviors

These types of behavior, however, are not the only ones found in ECEC. As previously explained in Chapter 4, ECEC behaviors are expressed through 3 different ways: through equations, through activity diagrams or through a list of activities. In the case of equations, it is mainly composed of a set of variables (attributes and parameters) and constants. Activity diagrams, on the other

hand, are a graphical representation composed of two main elements (nodes and edges). The decision node may also be seen as a primitive activity termed decide, where a condition is evaluated. Lastly, the list of activities (termed activity behaviors) is simply a list of primitive activities. These types of behaviors and their main elements are represented in Figure 5.3

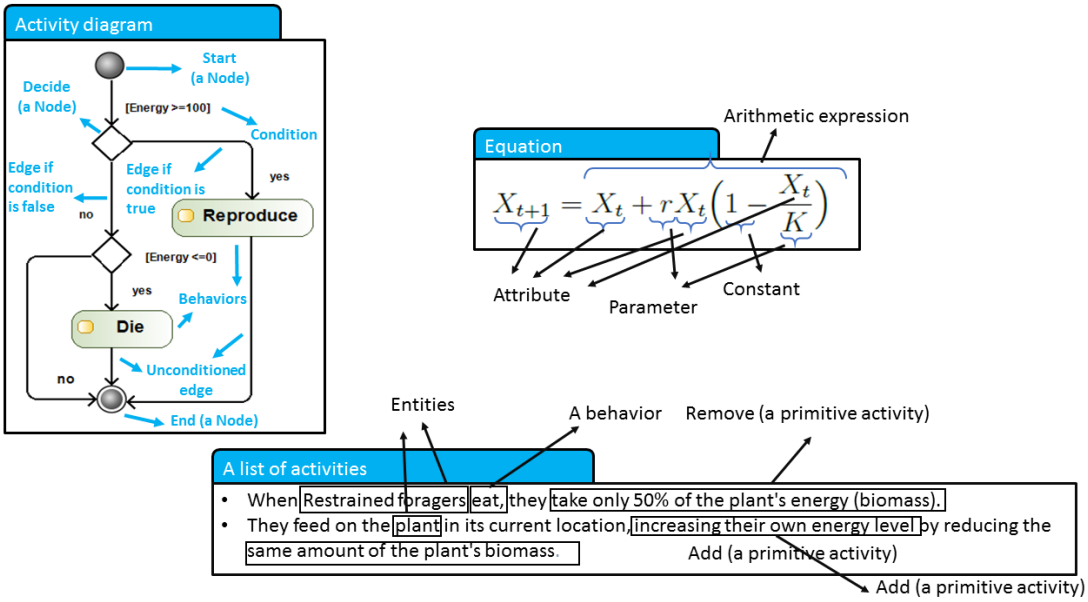


Figure 5.3 – Terms used by domain experts to model SES

Additionally, a natural language (such as the English Language) is used to describe behaviors: the combination of linguistic constructs (such as verbs and nouns) contains the same semantic elements for each behavior. This type of construction is presented in Figure 5.4, where we decomposed the structure of the "move" behavior of foragers. In this example, a set of locations is previously calculated (the neighbors of a location), and this set can be stored in a variable (e.g. local variables). Some expressions with verbs (such as "occupied by") represent a condition that is evaluated. Each condition can be represented by a Boolean function, with certain arguments (in this case, the location and the entity) and which that, naturally, is evaluated as true or false. The guard condition itself participates in the decision of an entity: the agent might "move" (a primitive activity) to a location or to another, depending on how the condition is evaluated. Therefore, this behavior can be formally represented as an activity diagram. The following figure describes the different elements of the forager movement in natural language.

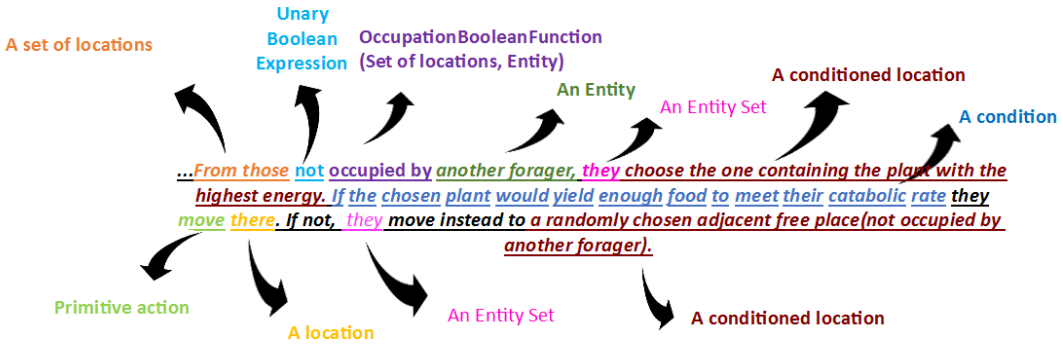


Figure 5.4 – An example of behavior described in natural language and its main elements

Based on these previously explained concepts and terms, we provide an informal semantic definition of DSL in Table 5.1.

Table 5.1 – B-Reactive's semantic domain

Terms	Semantics	
Model	A model is unique and contains descriptions of each agent and of their respective behaviors. It also contains definitions of the initial values of agents and the environment. A model must declare a main behavior, that, in turn, defines which behaviors will be performed by the model.	
Entity	An entity represents any agent in the model. Entities may have attributes and behaviors	
Behavior	A Behavior is associated to an entity. It represents what agents should do in the model. Behaviors are defined by optional set of parameters, local variables and expressions that are used to build the declaration of a behavior	
	Activity Behavior	A type of behavior that contains a sequence of primitive activities to be executed by the agent
	Activity Diagram Behavior	A type of behavior that provide elements (such as nodes and edges) to describe a behavior in a form of an UML Activity Diagram. Moreover, any behavior may also be seen as a node, and decision nodes can be used to express conditions.

	Equation Behavior	A type of behavior that represents all behaviors expressed in an form of equation. As in any type behavior, it can contain local variables, parameters and attributes. It must have an equation
Primitive Activities	Primitive activities are a type of pre-defined activities preformed by entities	
	Add	A primitive activity that adds a numeric value (represented by any type of variable) to an attribute.
	Remove	A primitive activity that removes a value (represented by any type of variable) from an attribute.
	Move	A primitive activity that returns a location to which an entity moves to. It is represented by primitive location (top, down, lef, right) or a location expression contained either in a variable, in an expression that returns a location
	Decide	A primitive activity that is used to specify the next action to be performed by an entity, based on the evaluation of a Boolean expression as true or false.
	Reproduce	A primitive activity that describes some entities reproduction parameters, such as the number of newborn siblings, the siblings initial position and the initial value of the attributes inherited by the the parent
	Die	A primitive activity that is used to express how an entity dies and disappears
Function	A function is a type of expression commonly used in the definition of a behavior. It reflects an action taken by an entity. A function has always an output, and may take some values as input. Both input and output values can be of String, Numeric, Boolean, Entity, EntitySet, Location or LocationSet types	

	Numeric function	All functions that return a numeric value. Basic arithmetic operation (such as plus, minus, multiplication and division) are example of numeric functions.
	String functions	All functions that return a string value. A function that returns the name of an entity is an example of such function. Concatenation is another example of such a function
	Boolean functions	All functions that return a boolean value. Comparison boolean functions (i.e. functions that compare two numeric values or two entities) are example of boolean functions
	Location functions	All functions that return a location value. An example is a function that returns the bottom location of a location.
	LocationSet functions	All functions that return a a set of locations value. For instance, the neighborhood of a entity returns a set of locations.
	Entity functions	All functions that return an Entity value. For example, a function that returns a random entity of the environment is Entity type function
	EntitySet functions	All functions that return a set of entities value. For instance, a function that returns entities having a certain value for their attribute is such a type of function
Initialization	The initialization represents the initial value of environment and the entities	
	Th environment's initial values	The initial value of the environment defines initial values (such as size) of the model's environment
	The entities initial value	The initial values of entities are defined by the act of initializing all entities that represent agents, as well as the parameters used by the entity's behaviors. It is also the part of model where the modeler instantiates all entities declared in the model

5.3 Abstract syntax for reactive behavior

The abstract syntax is usually represented by the use of trees. In that sense, abstract syntax trees (AST) are commonly used to represent the abstract syntax of natural languages, because trees are ideal for representing hierarchic structures. However, as stated by (Brambilla et al., 2012):

> *Meta-models define the abstract syntax of a language, but not the concrete notation of a language, such as graphical or textual elements used to render the models elements in modeling editors. This is a major difference between EBNF-based grammars which define the abstract syntax as well as the concrete syntax of textual languages all at once*

In MDE, a meta-model-centric design enables many concrete syntax (graphical and textual) to be developed for a modeling language. In that sense, we decided to use UML models to represent our abstract syntax (as a kind of MOF graphical representation).

5.3.1 The model

A model is the first element of the abstract syntax (or the top element if we consider an abstract syntax tree). It must have a name, at least one type of initialization (entities and environment), and it is composed by at least one entity. A model must also declare its main behavior, that is, what will be actually executed by the model's scheduler (i.e. a sequence of behaviors). A Model initializes entities and the environment and instantiates entities (see section 5.3.5 for more details about the model initialization). A model has also a scheduler, that is responsible for defining the sequence in which behaviors will be simulated. To that effect, the model's scheduler references at least one behavior, as shown in Figure 5.5

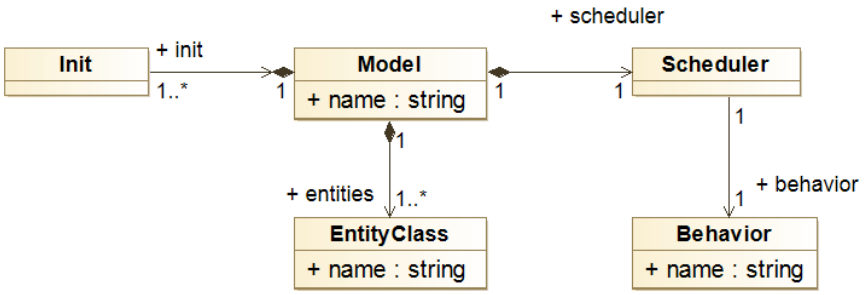


Figure 5.5 – The Model

5.3.2 The EntityClass

EntityClasses are part of a model and represent an Entity. An EntityClass must have a name and may have attributes (see section 5.3.4.1 for AttributeClass) and behaviors (see section 5.3.3 for Behaviors). An EntityClass is also located at a Location. The location has a neighborhood relationship with other locations. Considering a Moore neighborhood[1] a location's neighborhood should be composed by at least 3 other locations (in the case these locations represent the boundary of the space) and a maximum of 8 locations.

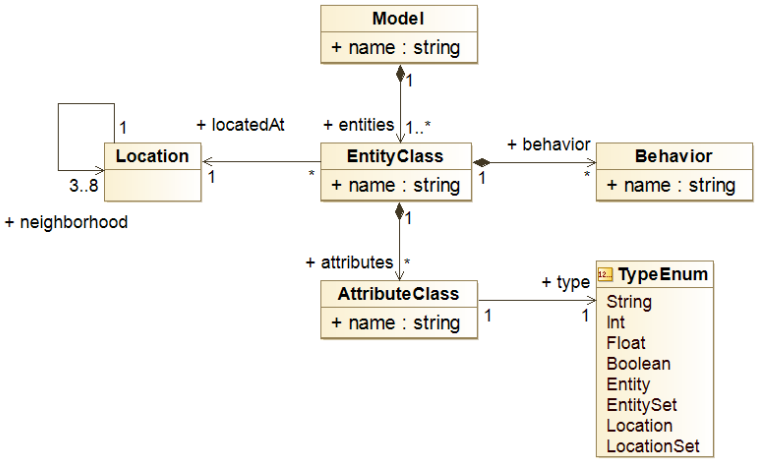


Figure 5.6 – EntityClass

5.3.3 Behaviors

Considering the behaviors explained in Chapter 4, three types of behaviors were included in the meta-model: ActivityBehavior, ActivityDiagramBehavior and EquationBehavior. Behaviors may have parameters and local variables and are types of variables (see section 5.3.4.1 for variable definition)

[1]The Moore neighborhood considers a two dimensional cellular space where each cell have eight surrounding other cells

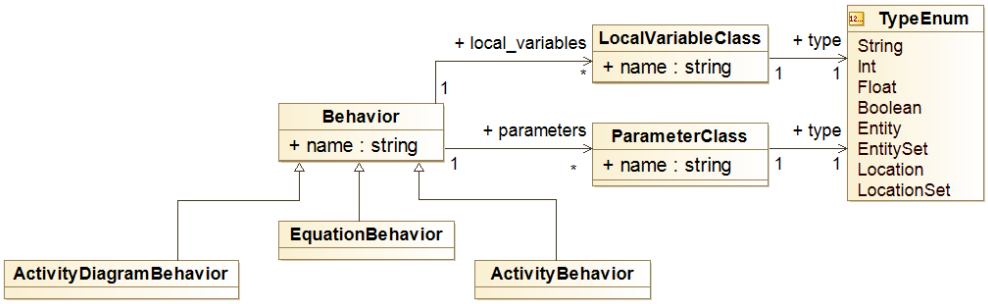


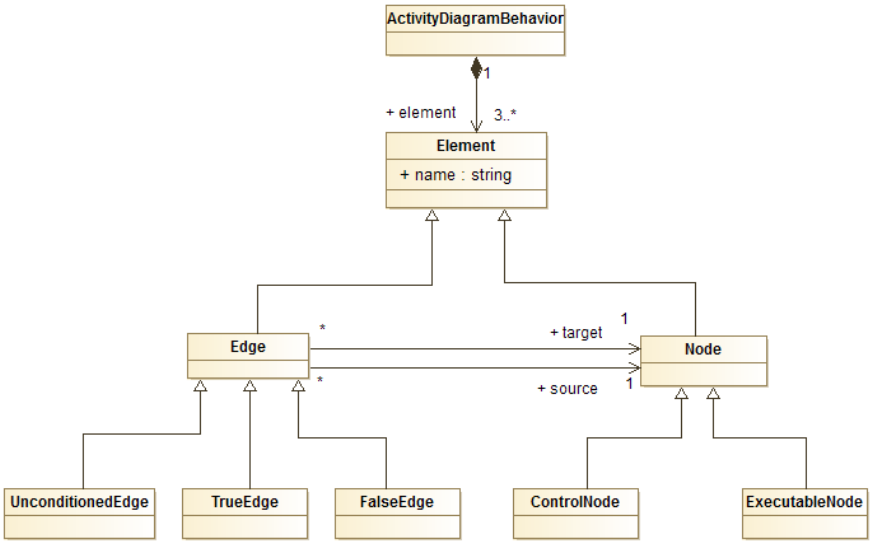
Figure 5.7 – Behavior

5.3.3.1 Activity Diagram Behaviors

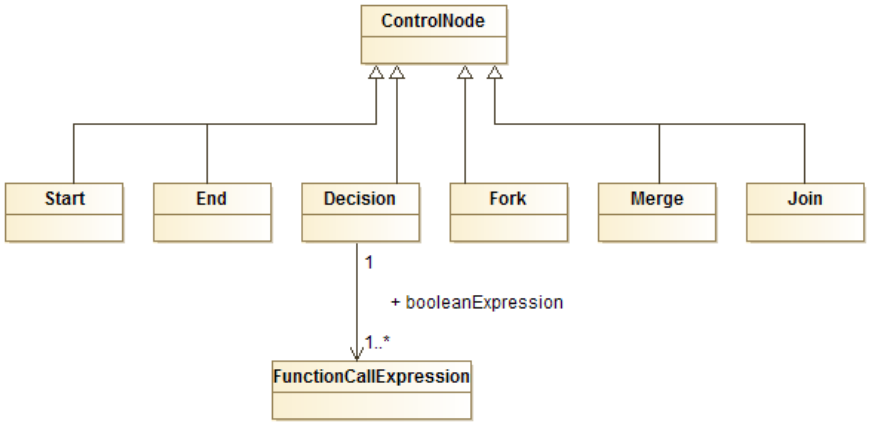
An ActivityDiagramBehavior class represents a subset of UML's activity diagrams. It is essentially composed of 2 type of elements: Nodes and Edges. Each Edge has two relationships with Node class: a many-to-one target relationship and a many-to-one source relationship. The Node element is specialized into ControlNode and ExecutableNode while possible types of an Edge element are: TrueEdge (for edges pointing to nodes when a Boolean function is evaluated as true), FalseEdge (for edges pointing to nodes when a Boolean function is evaluated as false) and UnconditionedEdge (for edges that simply connect one Node to another).

The ControlNode is a type node used to coordinate flows between other nodes. It includes the initial node (Start), the final node (End), the decision node (Decide) and the Merge, Fork and Join nodes. The Merge node brings together multiple incoming edges to accept a single outgoing flow. The Fork node has one incoming edge and multiple outgoing edges and is used to split an incoming flow into multiple concurrent flows. Finally, the Join node has multiple incoming edges and one outgoing edge, and is used to synchronize concurrent incoming flows.

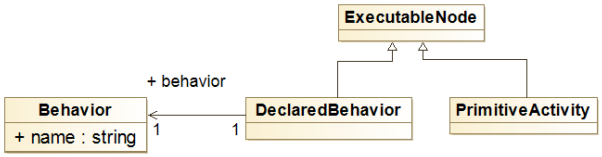
The ExecutableNode is a type of node used to represent an ordered arrangement of executable activities. They can be a DeclaredBehavior (a reference to a previously declared behavior) or a PrimitiveActivity (section 5.3.3.3)



(a) Activity Diagram Behavior



(b) ControlNode



(c) ExecutableNode

Figure 5.8 – Activity diagram behavior (a) and its subtype of nodes : ControlNode(b) and ExecutableNode(c)

5.3.3.2 Activity Behaviors

Activity Behaviors are composed by at least one primitive activity (section 5.3.3.3). Because of its sequential nature, the main difference between an ActivityBehavior and an ActivityDiagramBehavior is that no control nodes (like Start, Decision and End) are necessary. It is a simple type of behavior that is composed of a sequence of primitive activities.

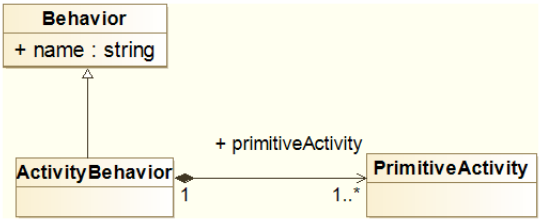


Figure 5.9 – ActivityBehavior

5.3.3.3 Primitive Activities

PrimitiveActivity is a type of Node that represents activities that cannot be decomposed. These activities can be specialized into Move, Add, Remove, Reproduce, Die, and Set (see Figure 5.10). Add and Remove activities refer to the action of adding or removing a value respectively to and from a numerical variable. Reproduce is a primitive activity with three relationships: a relationship to a Constant-Expression (representing the initial number of new population), a relationship to an Expression (representing the initial location of the newborn entities) and a relationship to a FunctionCall expression (representing the initial values of the newborn's entity attributes). The Set primitive activity is composed by an Assignment, allowing a new value to be assigned to a variable through an expression

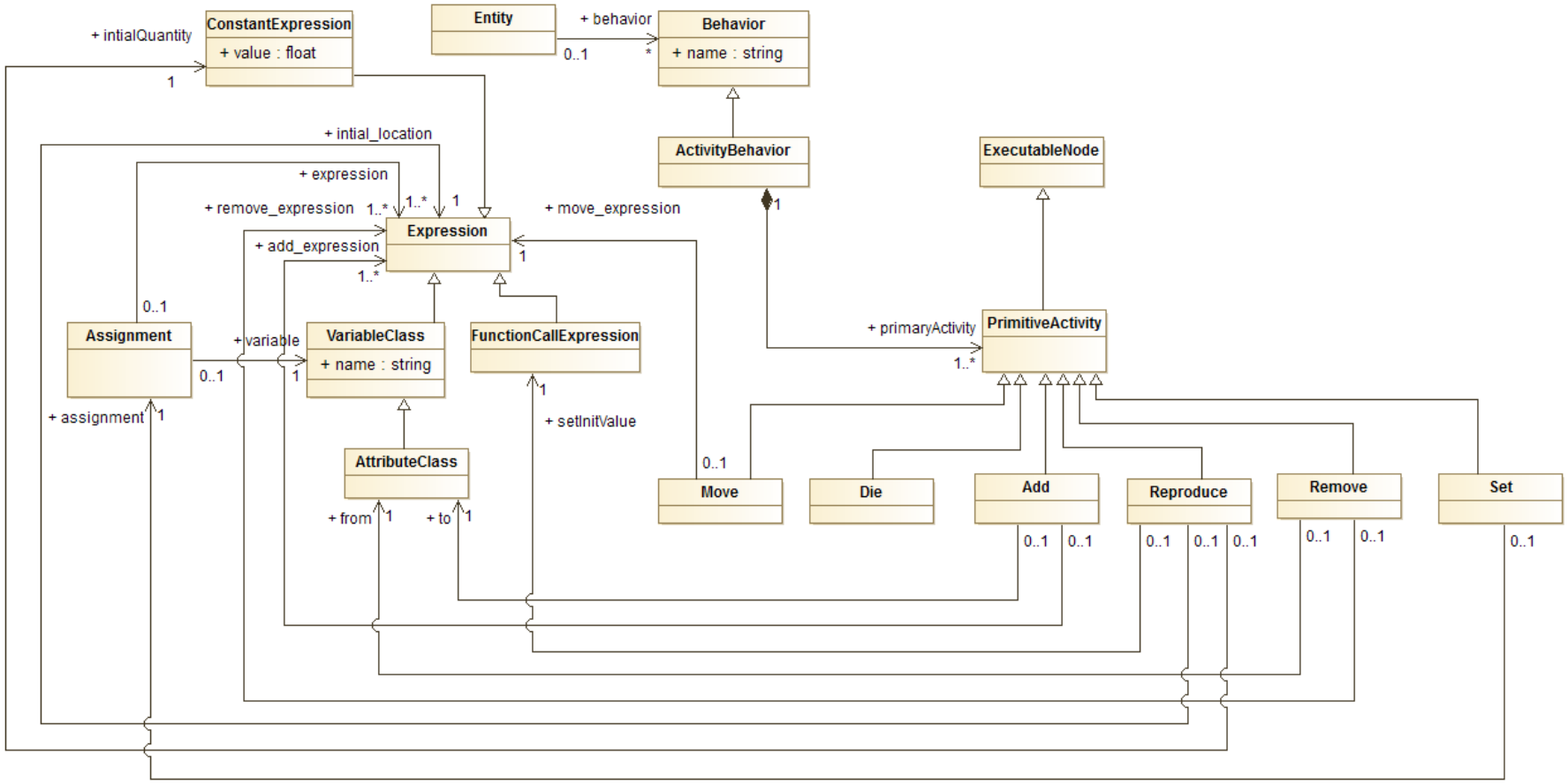


Figure 5.10 – Primitive Activity class diagram

5.3.3.4 Equation Behaviors

EquationBehavior is a behavior described by equations. An Equation is composed of LHS (short for left-hand side) and RHS (short for right-hand side). Therefore, LHS is an AttributeClass, and RHS represented by one expression.

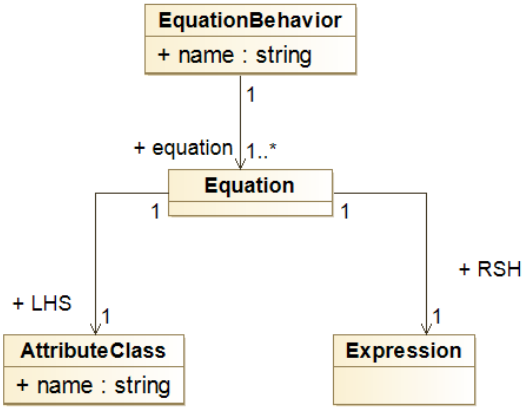


Figure 5.11 – ActivityDiagramBehavior

5.3.4 Expressions

The abstract syntax often expressed by the data structures of the language used for its implementation, instead of the syntax elements that are part of the concrete syntax. For that reason, some representations in the abstract syntax might differ from the ones used to represent the concrete syntax. That is the case of the expression in our abstract syntax. The expression is an abstract class with three subclasses: ConstantExpression (with an attribute value of float type), VariableClass (section 5.3.4.1) and FunctionCallExpression (section 5.3.4.2).

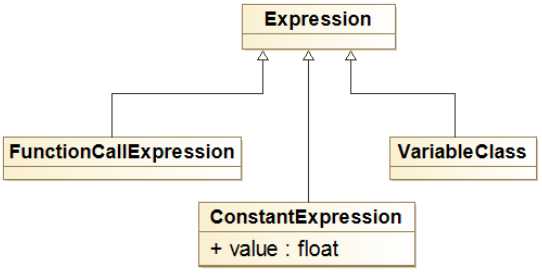


Figure 5.12 – Expression class diagram

5.3.4.1 Variable Class

Like expressions, VariableClass is an abstract class that represents variables. The VariableClass has one name, one type (e.g. String, Float, Boolean, Location, LocationSet, Entity or EntitySet) and it is specialized by each of its subclasses: AttributeClass, ParameterClass and LocalVariableClass.

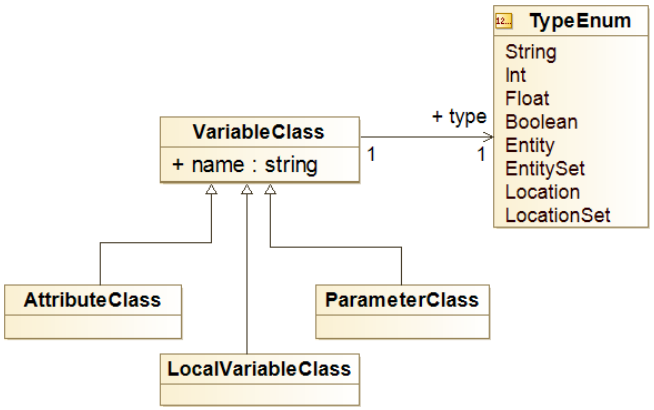


Figure 5.13 – VariableClass class diagram

5.3.4.2 Function Call Expressions

FunctionCallExpression is a type of expressions composed of at least one function. A function is a relation from a set of inputs to a set of one possible output, where each input is related to exactly one output. Its formal definition is given by $f : X \to Y$, where X is the set of inputs (called domain) and Y is the only output (called co-domain). To every domain and co-domain, a type (TypeEnum) is associated, being $\mathbb{S}$ for string type, $\mathbb{B}$ for boolean type, $\mathbb{N}$ for (integer or float type), $\mathbb{E}$ for Entity type, $\mathbb{ES}$ for EntitySet type, $\mathbb{L}$ for location type, and $\mathbb{LS}$ for LocationSet type. Functions can be of two types: a predefined named function, or an anonymous function. The latter uses parameters and an expression to define itself.

In that sense, the abstract syntax for function call expressions shown in Figure 5.14 provides all the necessary elements to implement each function implemented in the DSL. These functions are instances of the Function class, and since they belong to the concrete syntax representation of our DSL, we will provide more details about function call expressions in section 5.4.5

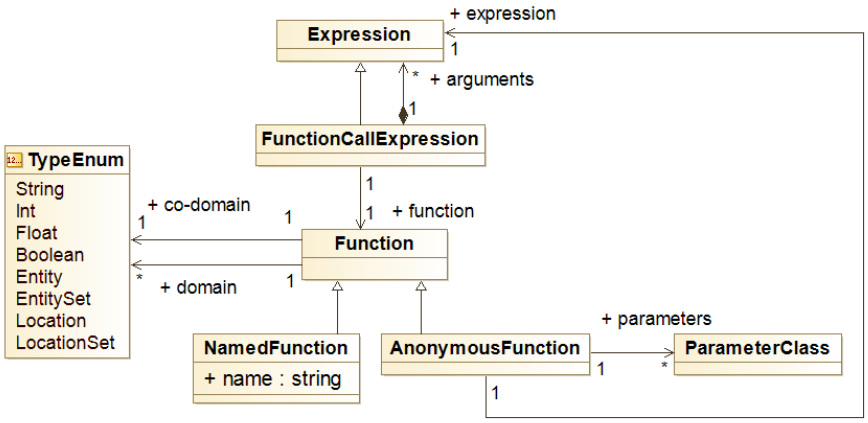


Figure 5.14 – FunctionCallExpression

5.3.5 Initialization

The Init class is an abstract class specialized in two subclasses: InitSpace and InitEntity. The first (InitSpace), has a "function call expression" to initialize the space. The second (InitEntity) has a function call to instantiate entities and another function call to define the initial position of Entities. It may have many assignments that define the initial values of an entity's attributes and parameters.

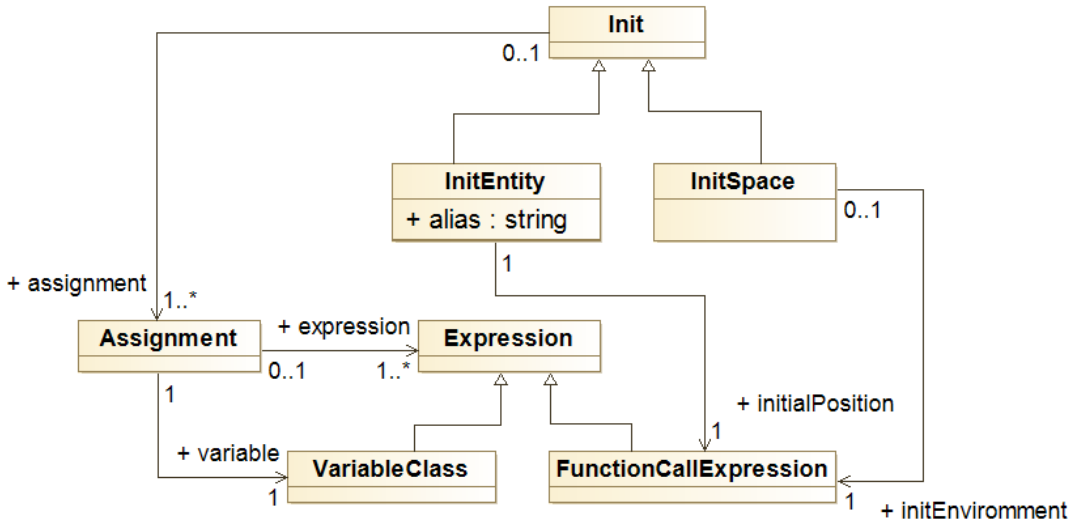


Figure 5.15 – Init

5.4 Concrete syntax

The advantage of UML is that it is a formalism that is independent of both graphical and textual concrete syntaxes. However, some notational elements do not have a corresponding abstract syntax element in UML (e.g. punctuation symbols to delineate tokens in textual concrete syntax). For that reason, concrete syntaxes are defined based on a grammar specification such as EBNF. A possible notation to express EBNF grammars is through syntax diagrams (also known as railroad diagrams). They are very often used to explain syntaxes in programing languages documentations [2]. Nevertheless, to facilitate the understanding of our concrete syntax, we will provide some syntax diagrams along with some examples that illustrate how B-reactive language can be applied.

5.4.1 A model declaration

An model specification starts with the keyword `Model`, followed by its name declaration represented by an ID. An ID is an identifier starting with a letter, followed by a sequence of letters, digits, or underscores. It is usually defined to describe names with or without numbers. Qualified names, in turn, are made of an ID preceded by an optional dot-separated sequence of identifiers. Qualified names are used to define a fully qualified name. In programming languages, a fully qualified name is an unambiguous name that specifies which object, function, or variable a call refers to, whatever the context of the call. More precisely, they are used to distinguish possibly similar names that are declared in a different scope of a model.

The Entities are declared later, followed by the scheduler definition, which, in turn, is represented by a behavior. The scheduler declaration starts by using the keyword `Run-main-as:`, followed by the name of a behavior. Next, entities and the environment must be initialized. The concrete syntax for the model initialization is showed in section 5.4.6. Figure 5.16 shows the syntax diagram for a model definition, and code 5.1 present how a model is declared in B-reactive language.

[2] Oracle MySQL (http://dev.mysql.com/doc/x-devapi-userguide/en/mysql-x-expressions-ebnf-definitions.html) and SQLite (https://www.sqlite.org/lang.html) documentation syntaxes for SQL are both expressed in RailRoad diagrams

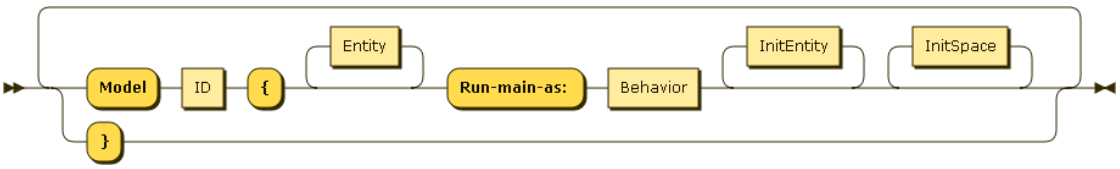


Figure 5.16 – Syntax diagram for a model definition

B-reactive syntax 5.1 – Model declaration

```
1 Model myModel {
2  Entities declarations{...}
3  Run-main-as : aDeclaredBehavior \\ Run the scheduler
4  Entities initialization and instantiation...\\
5  Environment initialization and instantiation...\\
6 }
```

5.4.2 The Entities declaration

Entities declaration starts with keyword `Entity`, followed by a name (ID rule) and brackets. Next, an entity may declare its own attributes (see section 5.4.3 for attributes) and the behaviors it can perform (session 5.4.4). The syntax diagram and the concrete syntax definition of for entity in B-reactive language are presented in Figure 5.17 and in code 5.9 respectively.

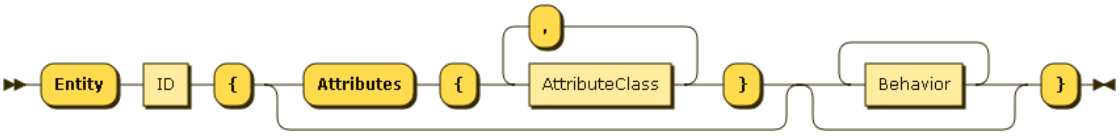


Figure 5.17 – Syntax diagram for an entity definition

B-reactive syntax 5.2 – Entity declaration

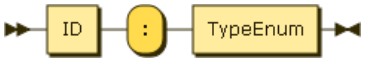
```
1 Entity anEntityName {
2  Attribute(s) declaration
3  ...
4  Behavior(s) declaration
5  ...
6 }
```

5.4.3 Attributes, Parameters and Local variables declaration

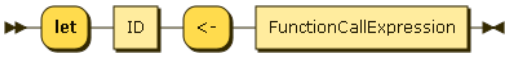
Attributes and parameters are initially declared by using keywords `Attributes` and `Parameters` (respectively), followed by a name (ID) and their type. Attributes can only be defined inside an entity definition (see previous section

5.4.2) and parameters, only inside a behavior definition (as shown in section 5.4.4). Like parameters, local variables, must be defined inside behaviors and its declaration starts with keyword `let`, followed an ID, followed by a function expression. The syntax diagram for attributes and parameters is shown in Figure 5.18a and local variable syntax diagram is represented in Figure 5.18b

In B-reactive language, attributes and parameters are defined according to the example presented in code 5.3



(a) Attributes and parameters



(b) Local variable

Figure 5.18 – Syntax diagram for attributes, parameters and local variable definition

B-reactive syntax 5.3 – Attributes and Parameters declaration

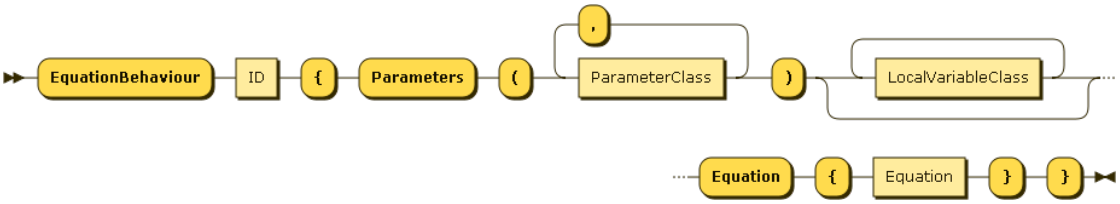
```
1  Entity Person {
2   Attributes {
3    color: String
4    size: Float
5  }
6  Behavior declaration.... {
7  Parameters {
8          distance: Float
9          time : Float
10         destination : Location
11 }
```

5.4.4 The Behavior declaration

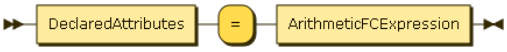
In B-reactive language, behaviors must always be defined inside entities declarations. Behaviors may have parameters and local variables. Like parameters, local variables declaration is done inside, but with a slightly different syntax: their definition starts with the keyword `let`, preceded by an ID, an assignment symbol, and an expression (see section 5.4.5 for expressions). Behaviors and can be of three types : Equation behaviors, Activity Behaviors, and Activity Diagram behaviors.

5.4.4.1 Equation Behavior

As shown in code 5.4, Equation behaviors start with the keyword `Equation-Behaviour` followed by an name (ID) and the declaration of parameters. Next, an equation must be provided. Its definition starts with the keyword `Equation` where its LHS is defined by an declared attribute, and its RHS is defined by an arithmetic (numeric) function call expression (see section 5.4.5).



(a) Equation Behavior



(b) Equation

Figure 5.19 – Syntax diagram for an equation behavior and its equation

B-reactive syntax 5.4 – Equation behavior definiton

```
1 EquationBehavior anEquationName {
2  Parameters {
3          x : float
4          y : float
5  }
6  Equation {
7          aDeclaredAttribute = (x + y) / x * x
8  }
9 }
```

5.4.4.2 Activity Behavior

Activity behavior definition starts with the keyword `ActivityBehavior`, followed by a provided ID, a list of possible parameters and local variable definitions. An activity behavior must later define at least one primitive activity (see Figure 5.10). The syntax diagram for the activity behavior is presented in Figure 5.20. Code 5.5 shows the concrete syntax of B-reactive language to define activity behaviors.

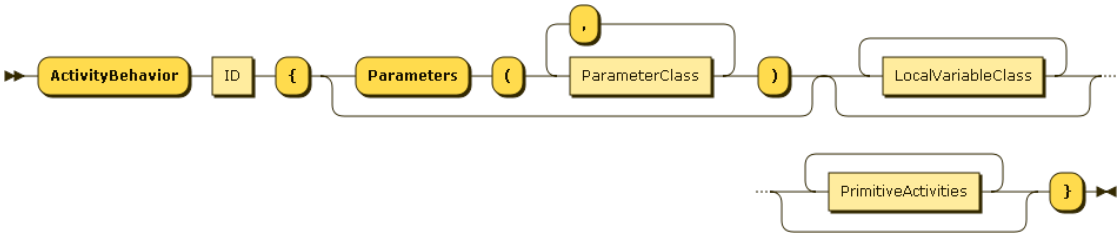


Figure 5.20 – Syntax diagram for the activity behavior

B-reactive syntax 5.5 – Activity Behavior definiton

```
1 ActivityBehavior Live {
2         Parameters {
3                 A List of parameters...
4         }
5         let aLocalVariable <- 5 \\A local variable definition
6         aList of primivite activities....}
```

5.4.4.3 Activity Diagram Behaviors

Although UML's Activity diagram is a graphical formalism, there are some reasons (according to Flater et al., 2009) to specify human-readable text form from a graphical formalism such as activity diagrams. First, graphical representations are very time-consuming to create and take a lot of space on a screen. The second reason is that a programmer can perform a faster prototyping when running human-readable text format for input or output since it takes significantly less effort and up-front investment. Additionally, many of the semantics elements contained in activity diagrams are similar to those of flow chart diagrams.

The declaration of an activity diagram behavior starts with the keyword `ActivityDiagramBehavior`, followed by a name, a list of parameters and local variable definitions and the initial rule (**Start**) representing the Start node. The syntax diagram for the activity diagram behavior is presented in Figure 5.21.

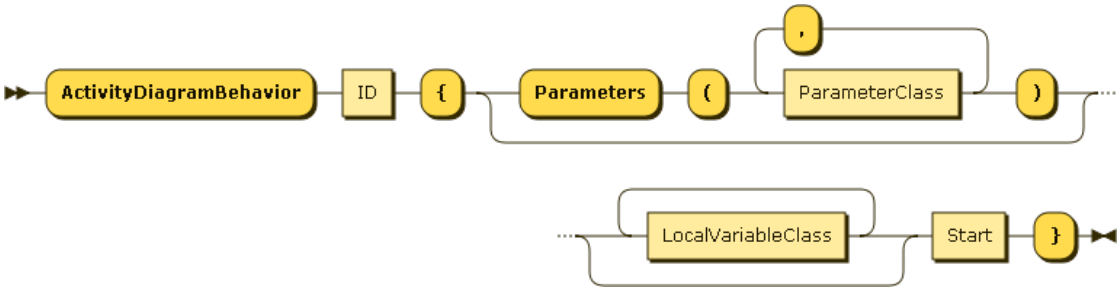
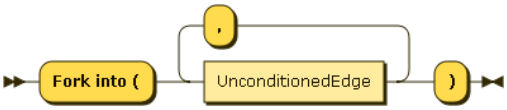


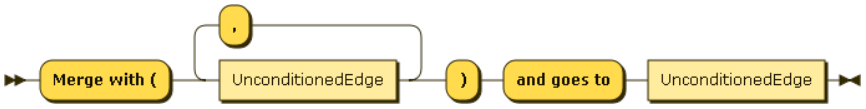
Figure 5.21 – Syntax diagram for the activity diagram behavior

The **Start** rule definition begins with the keyword Start, followed by the keyword -> that represents an **UnconditionedEdge** rule. Next, a node must be provided: it can either be a ControlNode or an ExecutableNode. Control nodes in turn can be Fork, Decide, Merge, Join or End. A syntax diagram for each control node is provided in Figure 5.22.

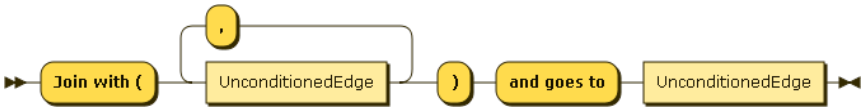
(a) Start



(b) Fork

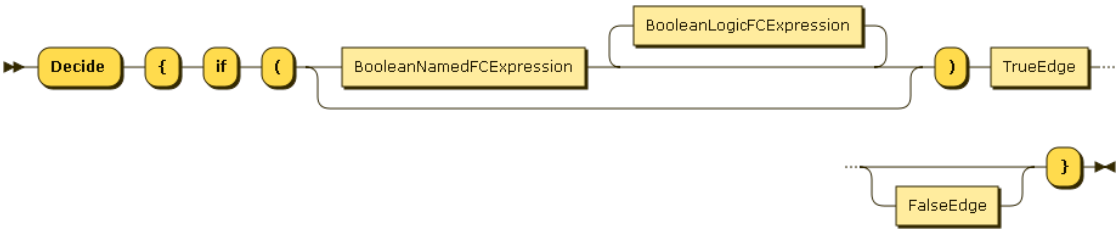


(c) Merge



(d) join

(e) End



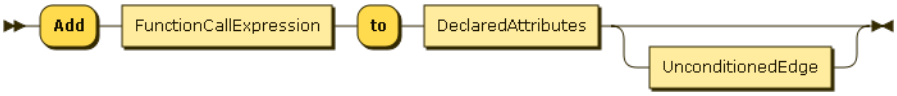
(f) Decision

Figure 5.22 – Syntax diagram for control nodes

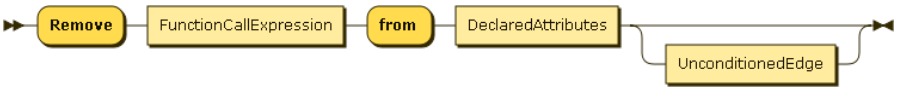
Executable nodes can be a previously declared behavior, or primitive activities. If the latter type is provided, then B-reactive language provides 6 types of primitive activities: **Add**, **Remove**, **Decide**, **Reproduce**, **Move**, **Die** and **Set**. **Add** and **Remove** are primitive activities starting with keywords `Add` and `Remove`, respectively. Next, a user inputed function call expression is calculated. The result of this calculus is either added `to` a declared attribute (if the primitive

activity is **Add**) or the result of the calculus is removed `to` a declared attribute (if the primitive activity is **Remove**). The primitive activity **Decide** can only be defined inside activity diagram behaviors. Its definition starts with the keyword `Decide` followed by the keyword `if` and a boolean expression (see section 5.4.5.5) that is evaluated, leading to another node. This node may be another primitive activity, a declared behavior, an `else` keyword (in case that the boolean expression is evaluated as false) or an `end` node that marks the end of a activity diagram behavior.

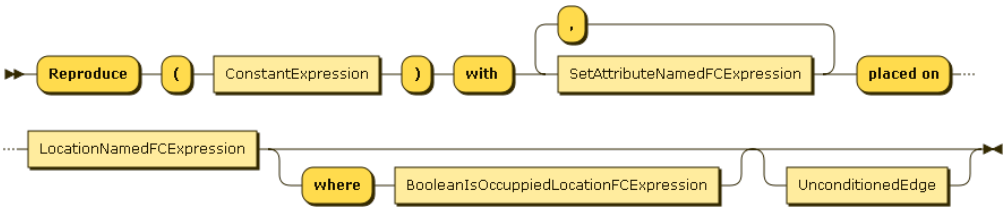
The primitive activity **Reproduce** starts its declaration with the keyword `Reproduce`. Its declaration is followed by a number that represents the initial number of the entity's decedents, a function expression (section 5.4.5) that sets an initial value of an attribute inherited by the entity, and an initial location given by a location expression (see section 5.4.5.1). The **Move** primitive activity starts its declaration with the keyword `Move-to`, followed by a location expression or a local variable that contains a location. The **Set** primitive activity starts with the keyword `Set` and it is in fact, an assignment, since it sets a new value to a declared attribute, or parameter. Finally, the **Die** primitive activity is simply defined by the use of the keyword `Die`. The syntax diagram for each primitive activity diagram is presented in Figure 5.6



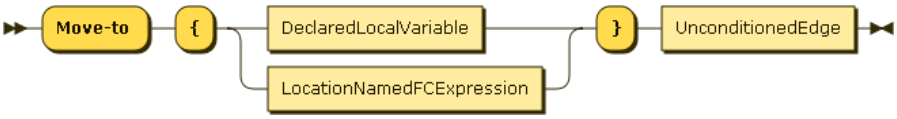
(a) Add



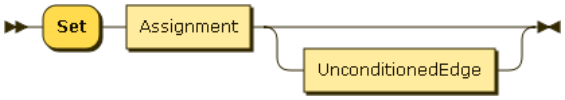
(b) Remove



(c) Reproduce



(d) Move



(e) Set

Figure 5.23 – Syantax diagram for primitive activities

To illustrate how an activity digram behavior is defined in B-reactive language, consider an example of a model with a single entity Parent, with attributes `age` and `maxlifetime`. The parent has two behaviors: `Live` and `WorkandReproducebehavior` that reproduces and move their children into any place that is occupied by any Parent. The activity diagram behavior (called `WorkandReproduce`) defines two parameters : `money` (as float type) and city (as a location type). In that example, we define a local variable called `aGoodCity`: its value is initialized with a random location (in the environment) that is not occupied by any Parent. Next, if the Parent is too old (that is, his age is equal to its `maxlifetime`), then he dies. Otherwise, if he is not too old (age is lower than `maxlifetime`) and `aGoodCity` contains at least one location, then the parent reproduces 1 child, with 10 initial units of money, immediately placing their children in a location given by the local variable `aGoodCity`. That algorithm is implemented with B-reactive language in code 5.6

B-reactive syntax 5.6 – Activity diagram behavior definition

```
Entity Parent {
 Attributes {
  maxlifetime: Integer
  age: Integer
 }
 ActivityBehavior Live {
  Parameters {
   money: Float
  }
  Add 1 to age
  Add 12.5 to money
  Remove 1 from maxlifetime
 }
}

ActivityDiagramBehavior WorkandReproduce {
 Parameters {
  money: Float
  city : location
 }

 let aGoodCity <- one-of city NOT occupied by Parent
 Start -> Decide { if (Live.age = Live.maxlifetime AND count(aGoodCity) > 0) ->
        Die -> End -> Else
         Reproduce(1) with money(10) placed-on aGoodCity ->
                Set Live.maxlifetime := Live.maxtimelife - 2 -> Live -> End
            } -> End
  }
}
```

5.4.5 Function Expressions

In B-reactive language, every expression that return values is a "function call expression". Some functions takes more than one argument or sometimes, takes no argument (i.e primitive functions). However, they must always return a type value: that value can be a location, a set of locations, an entity, a set of entities, a boolean, a numeric value or a string. These functions are usually associated to a name (different from anonymous functions) and classified according to their return type. Table 5.2 shows some examples of function call expressions, and in the next sections, we present the concrete syntax for some of these (and other) functions available in B-reactive language

Table 5.2 – Examples of functions according to their co-domain

Function Name	FunctionType	Definition
Here, Top, Bottom, Right, Left	Location	$f : \emptyset \rightarrow \mathbb{L}$
TopOf, BottomOf, RightOf, LeftOf	Location	$f : \mathbb{L} \rightarrow \mathbb{L}$
MaxOneOf	Location	$f : \mathbb{N} \times \mathbb{LS} \rightarrow \mathbb{B}$
GreaterThan, GreaterOrEqualThan, LessThan, LessOrEqualThan, Equal	Numeric	$f : \mathbb{N} \rightarrow \mathbb{B}$
Plus, Minus, Multiplication, Subtraction	Numeric	$f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
BooleanNumericComparison	Boolean	$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
Entities	EntitySet	$f : \mathbb{ES} \times \mathbb{ES} \times \mathbb{B} \rightarrow \mathbb{ES}$
IsOccupiedLocation	Boolean	$f : \mathbb{L} \times \mathbb{B} \times \mathbb{E} \rightarrow \mathbb{B}$
Not	Boolean	$f : \mathbb{B} \rightarrow \mathbb{B}$
And, Or	Boolean	$f : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
Oneof	Entity	$f : \mathbb{ES} \times \mathbb{B} \rightarrow \mathbb{E}$
SelectConditionedLocation	LocationSet	$f : \mathbb{L} \times \mathbb{B} \rightarrow \mathbb{LS}$
UnionLocation	LocationSet	$f : \mathbb{L} \times \mathbb{L} \rightarrow \mathbb{LS}$

5.4.5.1 Location Functions

Locations expressions (that is, expressions that return a location) can be primitives or non-primitives. Primitives locations takes no arguments and return a location. Considering the current location of the entity, primitive location functions `here`, `top`, `bottom`, `right`, `left` returns the the top, bottom, right and left position of this entity's current location, respectively

Other location expressions include `top-of`, `bottom-of`, `right-of` and `left-of`. In that case, they require a location as a parameter. Although the `one-of` is a $\mathbb{E}$ function that returns a random entity from a set of entities, it can also

can also be used to return a random location, given a set of The function. The function `max-one-of` returns a random location based on the maximum value of a attribute, given a set of locations as input. The function declaration consumes the **MaxOneOfLocationFunction** rule that in turn, returns the name of the function as a keyword `max-one-of`. Next, an declared attribute (a reference to an attribute) must be inputed as the first argument. The second argument should be a $\mathbb{LS}$ function (see section 5.4.5.2 for location set functions). The syntax diagram for `max-one-of` function is presented in Figure 5.24 and code 5.7 shows B-reactive concrete syntax examples of location functions.

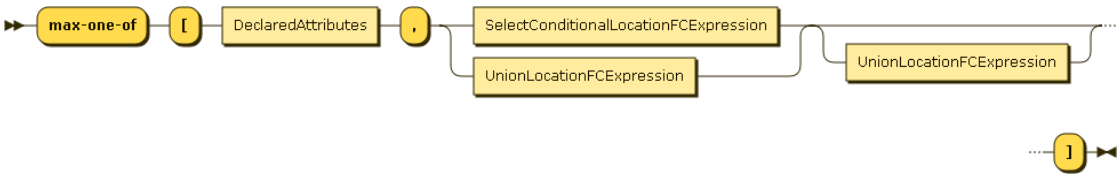


Figure 5.24 – Syntax diagram for the max-one-of location function

B-reactive syntax 5.7 – Location expression examples

```
1 top \\ Returns the top of an entity's current location
2 top-of here \\ Returns the top of an entity's current location (same as top)
3 left-of aLocation \\ Returns the left of "aLocation" variable
4 one-of aSetofLocation \\ Returns a random location from a set of locations
5 max-one-of [anEntity.anAttribute, aSetofLocations]
```

5.4.5.2 LocationSet Functions

These types of function expressions always return a set of location functions. Like $\mathbb{L}$ functions, $\mathbb{LS}$ functions also have primitives : `neighborhood` returns a set of locations that correspond the they neighbors of the entity's current location. Another primitive is `space`, that returns the environment of the models. As non-primitive examples of $\mathbb{LS}$ functions, the `union-location` function allows two or more locations to be grouped into a set of location.

CreateGridOf is also an example of $\mathbb{LS}$ function. It returns a set of locations using on a set of entities, the size of the grid, and an optional boolean expression. It is commonly used in the initialization of the model environment and its declaration starts with the keyword `create-grid-of`, followed by two arguments of type integer (representing the X and Y size of the grid) and a optional boolean express as an argument. See section 5.4.6 for more details about this function.

Another example of $\mathbb{LS}$ function is the **SelectConditionedLocation** function : it returns a location, based on set of locations and a given condition. This function is declared by using the keyword `select-location-from`, followed by a primitive location function ($\mathbb{L}$ or $\mathbb{LS}$), and ending its declaration with a condition (a boolean expression). Its syntax diagram representation is presented in Figure 5.25. Some few examples of B-reactive's concrete syntax of $\mathbb{LS}$ function are given in code 5.8

Figure 5.25 – Syntax diagram of SelectConditionedLocation function

B-reactive syntax 5.8 – LocationSet expression examples

```
1 one-of neighborhood \\ Returns a random neighbor
2 one-of space \\ Returns a random location of the environment
3 select-location-from aSetofLocations where (anEntity.anAttribute > 1)
```

5.4.5.3 Entity Functions

Entity functions returns only one entity. The **OneOfEntity** function is an example of $\mathbb{E}$ function. This function returns an entity from a given location and a $\mathbb{B}$ function (see section 5.4.5.5 for boolean functions). The **OneOfEntity** function declaration starts with keyword `one-of-entity`, preceded by a location (or a set of locations. This functions optionally a boolean expression, where the condition must be evauluated as `true`. If a $\mathbb{B}$ function is given, it must be preceded by the keyword `having`. The use of this function is illustrated in the syntax diagram 5.26.

To illustrate the use of this function in B-reactive language, consider the following example of a cat is placed in a environment with dogs of all races. Instinctively aware that bigger dogs are usually more quiet, the cat will seek a random place in the environment (represented by the landfield entity) where a dog's weight is greater than 10.5. The cats will move to a place that is occupied by random heavy dog. This example in B-reactive language, as shows code 5.9.

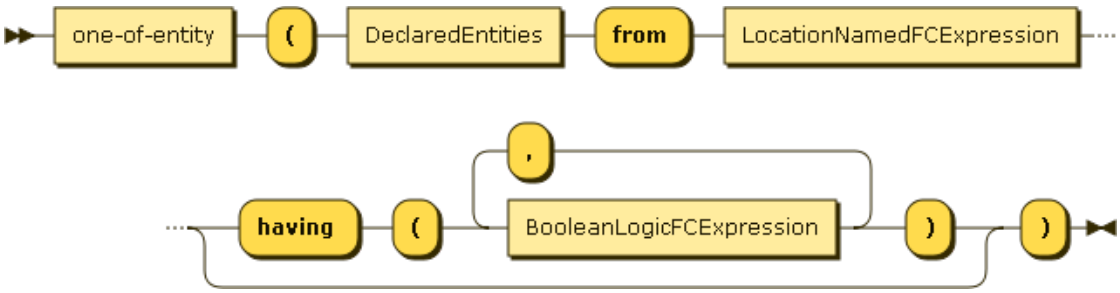


Figure 5.26 – Syntax diagram of function OneOfEntity

B-reactive syntax 5.9 – Entity function example

```
1  Entity landfield{
2  }
3  Entity Dog{
4   Attributes{
5    weight:Float
6   }
7  Entity Cat {
8   A behavior declaration...
9   \\ Returns an random heavy (and usually quiet) dog from the environment
10  let heavyDog <- one-of Dog from landfield having Dog.weight >= 10.5
11  \\ Returns a presumably safe location from the space
12  let aSafePlace <- select-location-from [space] such-that landfield
13  is occupied-by heavyDog
14  \\ Move to a safe place
15  Move-to aSafePlace
16 }
```

5.4.5.4 EntitySet Functions

Functions of $\mathbb{ES}$ type return a set of entities, such as the **Entities** function. This function returns a set of entities placed on the an primitive or a $\mathbb{LS}$ function, and that also satisfies a given boolean function. This function declaration takes three arguments : the first is the set of entities to be returned, preceded by the keyword `entities`. The second, is the $\mathbb{LS}$ primitive (or function), followed by the `having` plus the thrid and final argument of the function (boolean expression). The syntax digram for that function is represented in Figure 5.27 and an example of its concrete syntax in B-reactive language is implemented in code 5.10.

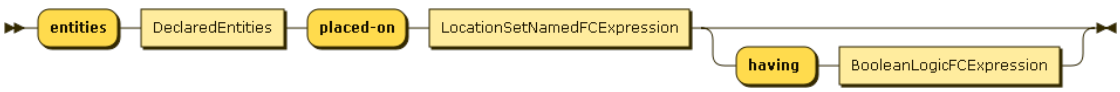


Figure 5.27 – Syntax diagram of Entities function

B-reactive syntax 5.10 – Entity set function example

```
Entity farmer {
 Attributes {
  type:String
 }
}
Entity landuse {
 Attributes {
  type : String
 }
}
Behavior declaration...
// Returns all the farmers placed on the pastures
entities farmer placed-on landuse
// Returns all secondary forests occupied by mechanized farmers
entities landuse placed-on landuse having landuse.type="secondary_forest"
```

5.4.5.5 Boolean Functions

The Boolean function always return a boolean value (`true` and `false`). Commonly, they can be logic (like `not`, `and`, and `or`), or comparison boolean functions (such as >=, <=, > and <). B-reactive also provides the location boolean function `IsOccupiedLocation`. That function returns a boolean value after evaluating if a location (location set) is occupied. The function definition starts with the the mandatory argument (a location or a location set) followed by keyword `is`. Next, a second optional argument may be prodivided (the unary boolean function `not`), followed by the keyword `occupied`. Finally, the third optional argument must be an entity or an entityset. If provided, it is preceded by the keyword `by`. Some examples are boolean functions in B-reactive language are presented in code 5.11

B-reactive syntax 5.11 – Boolean functions example

```
// Comparison boolean function
 let x <- 5
 let y <- false
 (x >= 4), (5 < 7), (6 >= 6), (y == false) // Returns true
 (x = 4), (5 > 10), (6 <= 5), (y != false) // Returns false
// Boolean location occupation
Entity Dog {...}
// Returns true if all locations around are occupied.
// Otherwise, returns false
let neighStatus <- neighborhood is occupied
// Returns true if all locations around are free.
// Otherwise, returns false
let neighStatus <- neighborhood is NOT occupied
// Returns true if all locations around not occupied by any dog.
let isNeighFreeOfDogs <- neighborhood is NOT occupied by Dog
```

5.4.5.6 Numeric Functions

The Numeric functions must return numeric values. They can be mathematic functions (such as `random-int`, `random-float`, `str`, `exp`, `power`, etc) or binary arithmetic functions, commonly used in equations (i.e. `+`, `-`, `/`, `-`). If we considering the set of numeric numbers $\mathbb{N}$, where $f : \mathbb{N} \rightarrow \mathbb{N}$ is $x \mapsto x$, then for every input(domain) x, and output (co-domain) x of same value is produced. For instance, if 5 is the input, then 5 is the output. Code 5.12 shows some example of numeric functions in B-reactive language.

B-reactive syntax 5.12 – Entity set function example

```
1 Entity cell {
2 Attributes {energy:Float}
3 Behavior declaration....
4 random-int 5, // Returns a random integer number between 0 and 5
5 2 power 5 // Returns 32
```

5.4.6 Model initialization

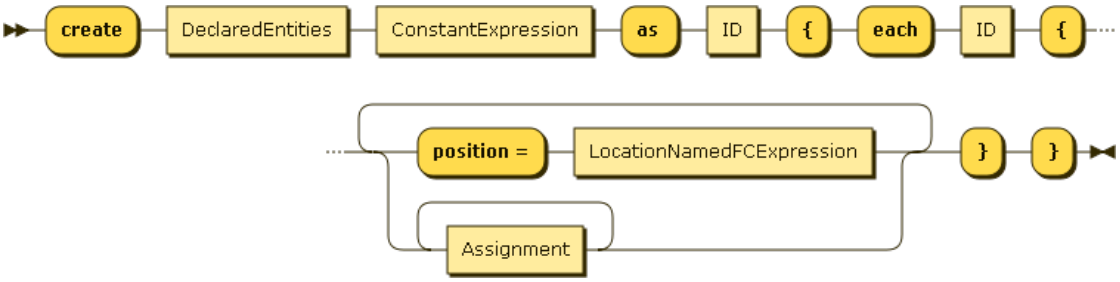
Due to the declarative notion of the B-reactive language, the behaviors of reactive agents are described, but they are neither instantiated, nor initialized. For instance, attribute and parameters of an equation (although valid), must have an initial value if they are going to be simulated. In MAS, located agents must declare a initial position in the environment The model must also instance a initial number of agents and define the environment's initial state. That can be perform through the rules **InitEntity** and **InitSpace** after the model is defined (as shown in section 5.4.1)

In our B-reactive syntax, the **InitEntity** rule allows agents to be instantiated and initialized. To instantiate agents, the function definition starts with the keyword `create`, followed by the name of the entity, followed by an ID that distinguishes eventual instances of the same agent type. That allows the simulation of heritage mechanism present in objected-oriented programing languages. We will discuss about that in more details in Chapter 6.

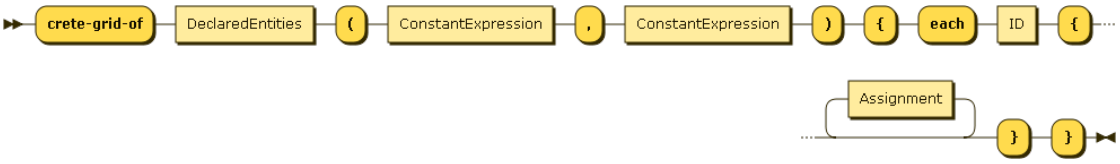
Next, to initialize agents, for every entity, an initial value is assigned (through the assignment rule) to every attribute that belongs to the entity. We must also provide the initial value to every parameter declared in entities' behaviors. Finally, the entity's initial position must be provided (see section 5.4.5.1 for location functions). The initial position declaration stats with the keyword *position*, succeeded by any functional call expression with a L function as co-domain. Its

syntax diagram is depicted in Figure 5.28a. An example of entities instantiation and initialization in presented in code 5.13, from lines 18 to 32.

The **InitSpace** rule enable the initialization of the environment. To instantiate the environment, the function `create-grid-of` (as explained in section 5.4.5.4) is used to create a grid of locations with a specif size. Later, the grid initializes its attributes values through assignment rules. The syntax diagram of **InitSpace** rule is presented in Figure 5.28b and the code implemented in Figure 5.13 shows, from lines 12 to 17, an example of the environment's instantiation and initialization.



(a) InitEntity



(b) InitSpace

Figure 5.28 – Syantax diagram for InitEntity and InitSpace rules

B-reactive syntax 5.13 – Model initialization

```
1  Model aModel {
2   Entity soil{
3    Attributes {
4     type:String
5     fertility:String }
6   }
7   Entity villager {
8    Attributes {
9     activity:"String"
10    incoming:"Float" }
11  }
12 // Instantiate and initialize the environment
13 create-grid-of soil (15,15) {
14  each soil {
15   type:= "terra firme"
16   fertility:="poor" }
17  }
18  // Instantiate, initialize and place 15 villagers as big farmers
19 create villager 15 as big-farmers {
20  each farmer {
```

```
21    position = one-of [grid-of Soil]
22    activity: := "agriculture"
23    income: := "50.5" }
24   }
25  // Instantiate, initialize and place 15 villagers as small farmers
26  create villager 15 as small-farmers {
27   each farmer {
28    position = one-of [grid-of Soil]
29    activity: := "artisan"
30    income: := "32.6"      }
31   }
32  }
```

5.4.7 Conclusion

In this chapter, we presented the implementation of the three main steps (abstract syntax, semantic domain and concrete syntax) of the design process of our B-Reactive language. B-Reactive is a DSL designed to specify reactive behaviors for MAS. Its design process was realized thought a meta-modeling approach based on existent concepts proposed in a simple SES model. From this simple model, we were able to specify an initial concrete syntax that is closer to the discourse used by stakeholders and SES non-programmers. The language is platform-independent and oriented towards natural speech. As in all DSL, the language is very prone to modifications. However, through the meta-modeling approach, the concrete language may by easily extended by adding new functions and adapting the concrete syntax to the domain of experts with no (or very few) changes in the abstract syntax.

Chapter 6

Implementation of B-Reactive language using MDE

6.1 Introduction

In this chapter, we discuss how we applied MDE based on a DSL, to develop a textual editor for B-reactive language (Chapter 5). We propose an approach based on 5 steps: 1) the implementation of a textual editor, 2) the development of language validation rules, 3) abstract syntax analysis of the target language, 4) code generators development and 5) test and validation of generated code. That approach is introduced in section 6.2. In section 6.3 we discuss the possible approaches (and advocate our choice) of implementing a textual editor and in section 6.3.0.2 we demonstrate how (and why) we added some validation rules to the editor. Section 6.5 is used to demonstrate how we implemented code generation into two MAS target platforms (Cormas and Netlogo) using MDE model-to-text capabilities. Next, we use section 6.6 to briefly discuss some simulation results from the generated code of two SES models (ECEC and Prison Rebellion) implemented in B-Reactive language. Finally, session 6.7 concludes this chapter

6.2 Application of MDE

We apply MDE by following a cyclical process of activities (Figure 6.1). In the first step, a first version of a textual editor was implemented using B-reactive concrete syntax specification. Later, we added some validation rules to cover certain constraints that ensure model's validity. In the next step, we analyzed the abstract syntax of two target MAS platforms (Netlogo and Cormas). This

is an essential step if we aim to build code generators for 2 or more target languages, since each target language requires a different transformation strategy. Based on that, the next step was to develop two code generators for the two target platforms (Netlogo and Cormas). To validate and test code generators, two SES models were implemented in B-reactive language, and source code for Netlogo and Cormas platforms were generated. The textual editor is later modified (grammar rules) and the process re-started

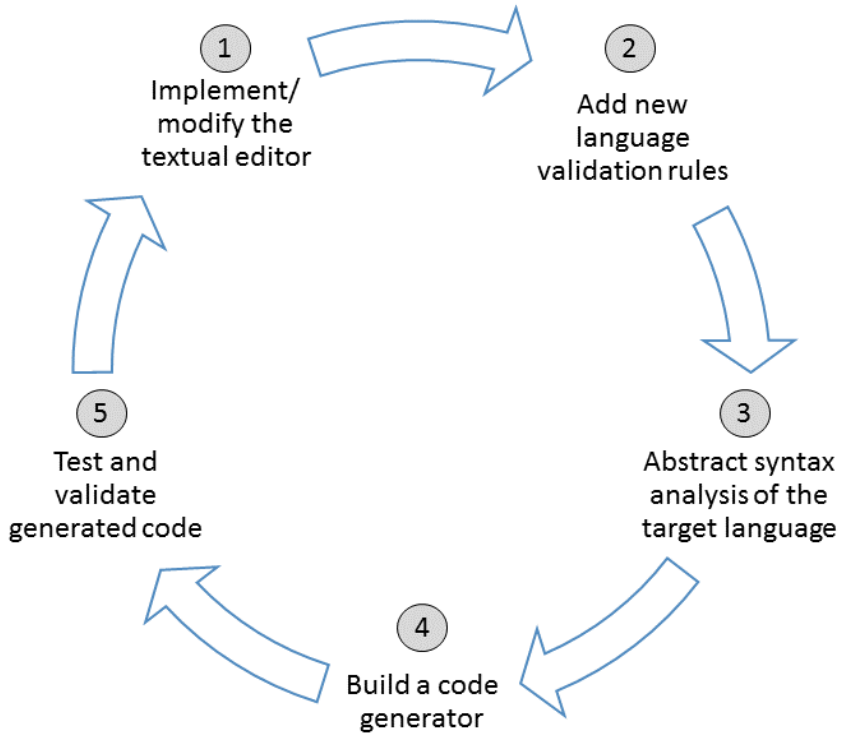


Figure 6.1 – Cyclical process of MDE application to obtain a DSl textual editor with code generator capabilities

6.3 Implementing a textual editor with XText

In order to implement DSL textual editor, (Brambilla et al., 2012) states that two possible approaches can be employed: a Grammar First approach or a Metamodel First approach. We identified a third possible approach if Xtext is chosen as a textual concrete syntax framework. Based on the authors' classification and our experience, the three known approaches are:

1. **Grammar centered approach** : The grammar is more important : The syntax is specified by the grammar (abstract and concrete syntax) and the

meta-model is not a major concern. A meta-model is later derived from the produced grammar;

2. **Meta-model centered approach** : The metamodel is more important. The meta-model is first specified, followed the concrete syntax specification that is conform to the meta-model;

3. **Meta-model and grammar-centered approach** : Both the meta-model and grammar are important. They are independently specified and the produced meta-model that is automatically inferred by the grammar is later transformed according to a model-to-model specification through the use of an M2M language

Approach number one (1) is by far the easiest and the most recommended way of working with Xtext. It is also the fastest since a previous meta-model is not required. However, this might not be the best approach if we aim to perform code generation into two or more target languages. The reason is that, although Xtext has good meta-model inference capabilities, some customizations to the inferred Ecore meta-model are very often required (e.g. when adding a derived operation). In that case, two options are left: 1) using some customized post processing rules; or 2) moving to a manually maintained meta-model. The first option requires much code adjustment (and knowledge) of the many core classes of Xtext. The second option leads to approach number two (2), which uses a manual meta-model as an input for the development of the textual concrete syntax.

Approach number two (2) is very often used by the archived TCS project, considers the meta-model as the first and most import element of the grammar. Xtext also supports this type of approach, but unfortunately, there is not much documentation if the toolsmith decides to build his grammar using a meta model centered approach. Despite this, no meta-model is generated, but models created are fully conform to the manually maintained Ecore meta-model

Nonetheless, Xtext syntax must be considered when using this approach. Xtext grammar language is not exactly EBNF but very similar. It extends EBNF by adding object-oriented concepts and the information necessary to derive meta-models and modeling editors. This can be considered an advantage from the point of view of grammar language flexibility or meta-model inference capabilities. But, since the meta-model is manually maintained, Xtext syntax requires return types (Ecore EClasses) for each rule specified in the grammar. In addition, possible left factoring or left recursion might occur and although this can

be solved by adding new production rules, it may create some ambiguities in the grammar. But because the rules must return a type (EClass) previously defined in the input meta-model, using the meta-model centered approach may lead to some loss in grammar flexibility. If, however, the toolsmith wants to keep models strictly conform to a manual provided meta-model, that is definitely the approach to go with.

Finally, approach number three (3) is available for toolsmiths desiring to keep some flexibility in grammar development but who still want to keep an externally maintained meta-model. In larger projects such as B-Reactive, the inferred meta-model is considerably different from the meta-model that is manually maintained. The reason is that additional rules are sometimes necessary (especially when solving left recursivity). But since the Ecore inferred meta-model translates the specified grammar elements into a Ecore model, an Ecore EPackage, for instance, is created for each generate declaration (Xtex syntax); an EClass is created for each return type of a parser rule (rules usually have a return type in Xtext syntax); an EEnum for each return type of an enumeration rule; an EDataType for each return type of a terminal rule or a data type rule, and so on.

To overcome this difference in models (conform to different meta-models), one option is to use a M2M language (like QVT) to transform the Xtext inferred model into a model that conforms to the manually maintained meta-model. One of the major advantages of using this approach is the high flexibility for meta-model and grammar specification. However, transformations may become quite complex (if not impossible) since the target implementation tends to be more complex due to added implementation details (Andova et al., 2012)

Despite all the drawbacks discussed (and advantages of other approaches), we chose approach number 2. Because our meta-model was used to describe our abstract syntax and because it was conceptualized with domain experts, a meta-model centered approach seems the most logical approach. Moreover, it is less time consuming compared to approach number 3, since an extra M2M transformation is not required. Our aim was to translate our meta-model (developed in Modelio modeling environment) into an Ecore meta-model. We shall discuss that step in session 6.3.0.1

6.3.0.1 UML to Ecore

In order to use our meta-model in EMF, the Ecore format is expected as a meta-model specification format. We used Modelio modeling environment to design B-reactive meta-model; Modelio offers two format possibilities when exporting a uml project: an uml file or an xmi format. Both formats may conform to UML standards (from version 2.1 to 2.4) or conform to EMF UML 3.0 standards

Although mapping from Ecore to UML is simple and direct (supported by the native EcoreTools editor), doing the opposite is not really possible. One reason is that although there's a fairly strong correlation among the 'type' concepts in Ecore and UML (for example, EClass, EAttribute, EDataType, EReference, EOperation, etc.) there are still some semantic mappings that needs to be solved.

Papyrus modeling environment supports direct export from a UML model into a Ecore model. But the whole meta-model must be initially designed in Papyrus. Hence, we decided to manually port our UML meta-model using the EcoreTools graphical designer provided by EMF.

6.3.0.2 Add new validation rules

Once the meta-model is the Ecore format, developing a textual editor with Xtext is a very straight forward process. The grammar syntax is very similar to EBNF and once the concrete syntax is implemented, a textual editor is automatically generated by the Xtext framework, as shown in Figure 6.2. Among many features, the generated editor provides auto-completion, highlight syntax and customizable features.

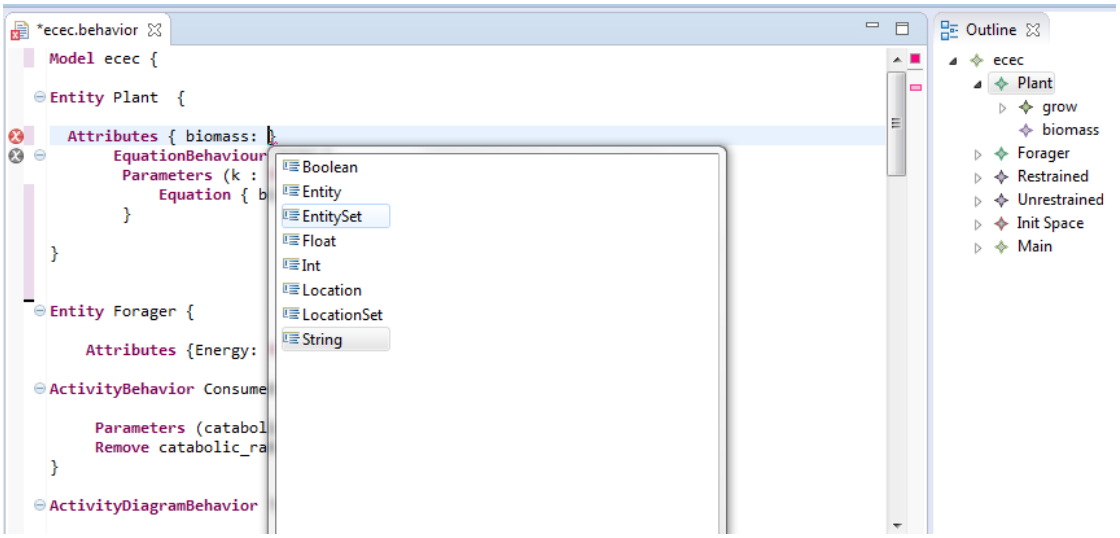


Figure 6.2 – A dsl textual editor generated by Xtext

Nonetheless, one of the key steps of modeling is model validation, where model's semantics are validated through the use of constraints. In EMF, OCL constraints can be added to the Ecore meta-model, using the OCLinEcore editor. The editor provides direct editing of Ecore meta-model, where the toolsmith may define constraints and invariants. Once defined, the constraints can be automatically evaluated by the textual (or graphical) editor. Figure 6.3 illustrates a constraint evaluation in the textual editor during the modeling process.

The OCL constraint imposes a restriction on all types of behaviors, where the name of behaviors must be unique (the violation message is shown on the bottom of figure. Even if at first sight this mechanism may seem well suited to a textual editor, the constraint message violation is not so user friendly . Moreover, validation checking is not so efficiently handled by the editor: when a constraint is violated, no fix solutions are offered to the user.

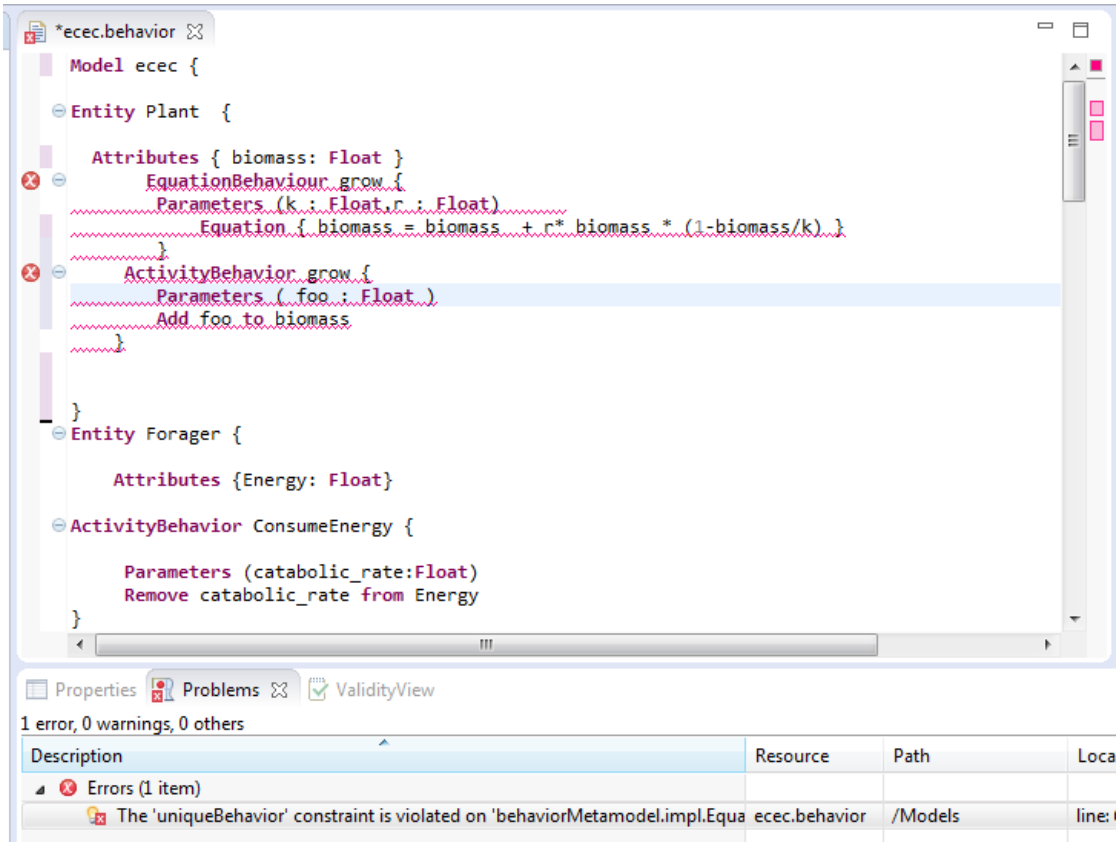


Figure 6.3 – A textual editor containing a message of OCL constraint violation

Nonetheless, one of the benefits of EMF environment is the possibility of implementing validation and quick fixes. Xtext provides default and custom validators that, through a serialization mechanism, convert the changes in the AST (executed by the quick fixes) back to text. More precisely, those validators provide `@Check` methods (for validation) and `@Fix` methods (for modification) definition to be directly applied in the Xtext editor. Modifications are implemented using quick fixes - a proposal to solve a problem in a program, Quick fixes are tightly connected to validation and act a semantic modification to the EMF resource. Although pure java can be used to customize DSLs, Xtext advocates the use of Xtend to specify validators. As mentioned in Chapter 3, Xtend is java-like programming language that has, among other features, a more compact and easier to use syntax.

With Xtext, we implemented some quick fixes for their correspondent warnings. Warnings are normally produced when a validation rule is not respected. Although errors may also have quick fixes, there is a consensus in most of modern editors to distinguish errors from warning. In that case, an error only detects (and points to) a customized error message, while warnings should have a quickfix. Figure 6.4 shows the editor after some quick fixes were implemented.

A validation rule that ensures that every entity's attribute name must start with non-capital letters triggers a quick fix that provides the user with a possibility of fixing warnings. Figure 6.4 shows a quick fix that automatically uncapitalizes the first letter of an entity's attribute name.

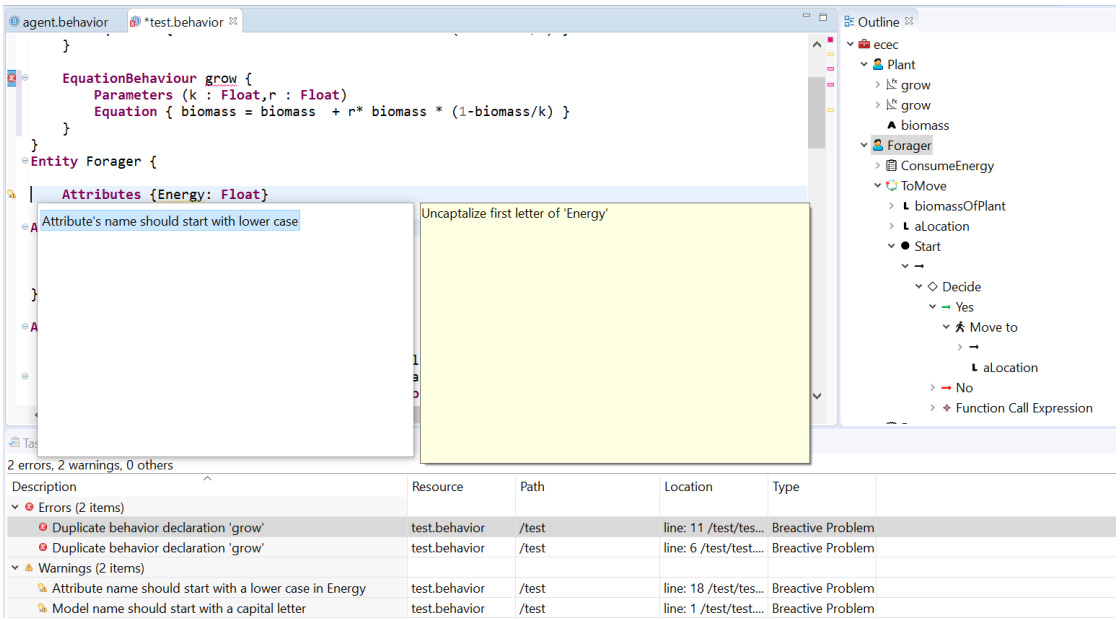
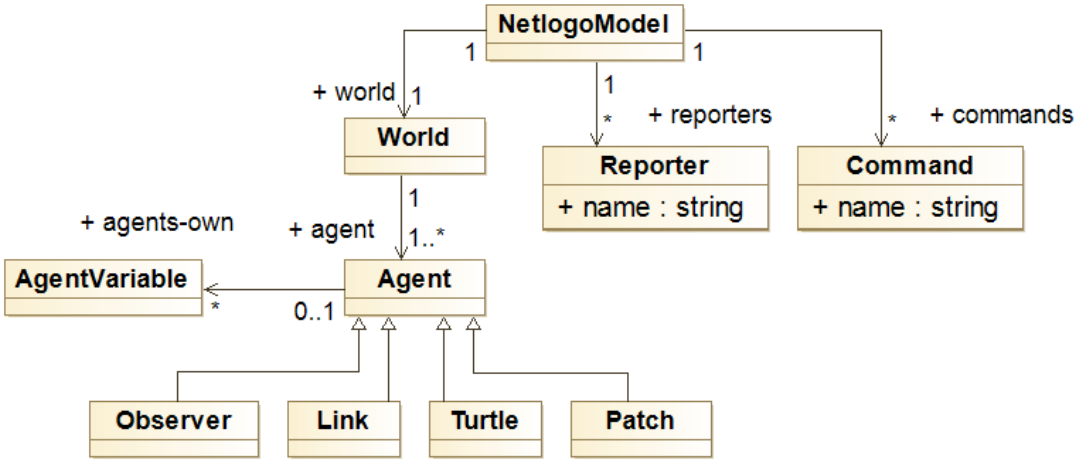


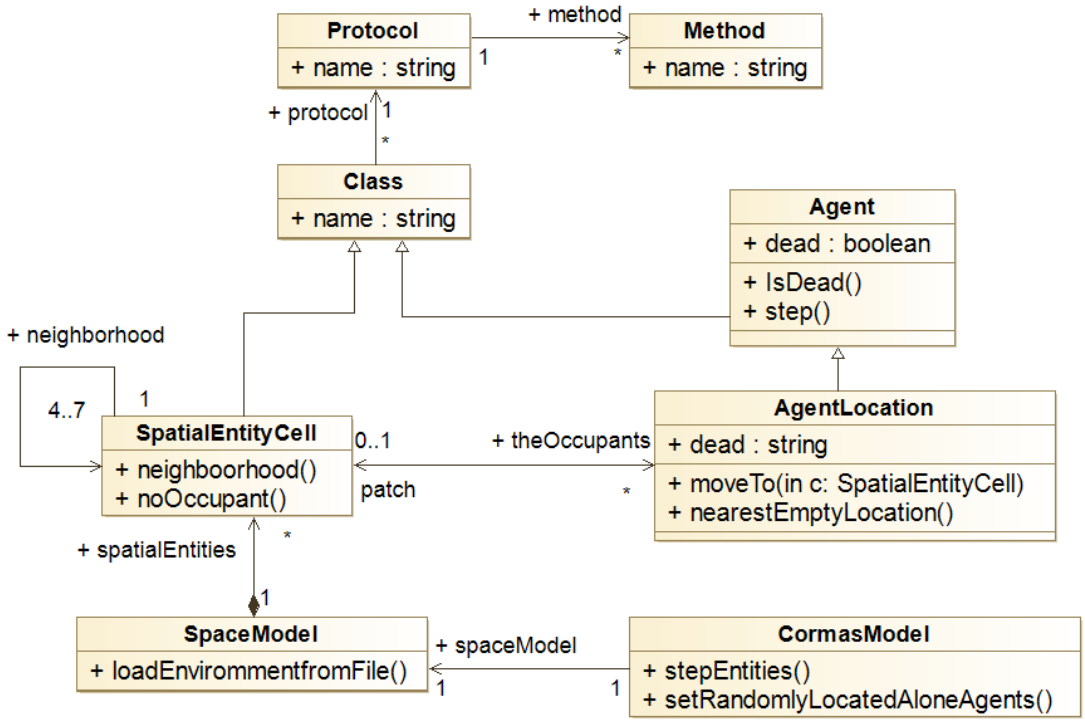
Figure 6.4 – B-reactive editor after implementation of validation rules

6.4 Abstract syntax analysis of the target language

In order to implement code generators for an specif MAS platform, we first need to define the abstract syntax for each of the target platforms. Although our aim is to build code generators for Netlogo and Cormas, it is essential to aim at what in fact we want to generate. In that case, we emphasize the behavioral aspect of these two platforms. However, because each platform has its own specificities (and different programing language paradigms), code generators had to be organized according to a general structure of how models are defined in each platform. Figure 6.5 illustrates this general structure for both Netlogo and Cormas platforms.



(a) Netlogo



(b) Cormas

Figure 6.5 – General abstract syntax for Cormas and Netlogo models

Netlogo models are usually composed of five type of agents: turtles (to represent agents), patches (to represent an environment unit, such as a cell, or land, for instance), links (connects two turtles) and the observer (oversees everything that's going on and does whatever the turtles, patches and links cannot do by themselves). Agents may have their own attributes. Netlogo also has commands and reporters that are used to tell what agents must do and to define a Netlogo's scheduler We will discuss more about them in subsection 6.6.

In general, models in Cormas are defined as follows: the agents and the environment are represented by Classes where their attributes and methods are inherited by other agents. That means that agents may inherit from other agents. To create agents and the environment, Cormas defines the `AgentLocation` class and `SpatialEntityCell` class, respectively. Every agent has a `step()` method : this method is executed at each time step of simulation. `SpatialEntityCell` also possesses its own methods, and inherits many others from its superclass `SpatialEntityElement` that in turn, is a subclass of `SpatialEntity` class. Both `SpatialEntityCell` and `AgentLocation` inherit many other methods from their correspondent super classes and for that reason, the abstract syntax presented in 6.5 is only partially presented.

Next, to initialize the simulation in Cormas, the agents and environment must be instantiated and the links between the entities, established. Cormas provides methods for scheduling (Scheduler) and for observation (Probes) of the simulation. Cormas classes may have protocols and class methods are defined inside protocols. We shall discuss Cormas methods (actually Smalltalk methods) in details in section 6.5.2

6.5 Building code generators

IIn the context of a M2T project, five template base languages are available to perform model-to-text transformation, but only two are mainly used: Acceleo and Xtend. Xtend is indeed a very powerful and easy-to-learn programming language. It can also be used for M2M transformation, model validation and test. But because of its full OCL support for querying models and its mature tool support, Acceleo was chosen, in this work,as the M2T language to implement code generation. The fact that Acceleo is based on templates (and not on code) is another reason for this choice: Templates can be more easily changed and since there is no programming language involved (such as Xtend), building tests or deploys are not necessary.

In order to demonstrate how our proposed language and the editor could be used to express models, we implemented two SES models using B-Reactive model: the Ecec model (as explained in Chapter 4) and the Prison rebellion model, a model based on (Wilensky, 2004) which in turn, is an adaptation of a model for civil violence proposed by (Epstein, 2002). Next, to test the Acceleo M2T code generation we implemented two different code generators: one aiming at

code generation for Cormas and the other, at NetLogo platform. Because each of those platforms have their particularities, code generators must be separately designed to support specificities possessed by each platform.

6.5.1 Code generation of Netlogo procedures

Once the general structure of Netlogo platform is defined, a set of code generator templates was implemented based on that platform structure. The next step was to specify a sub-structure for what we want to generate. Because we wish to generate procedures, an abstract syntax of how procedures are defined in Netlogo is shown in Figure 6.6

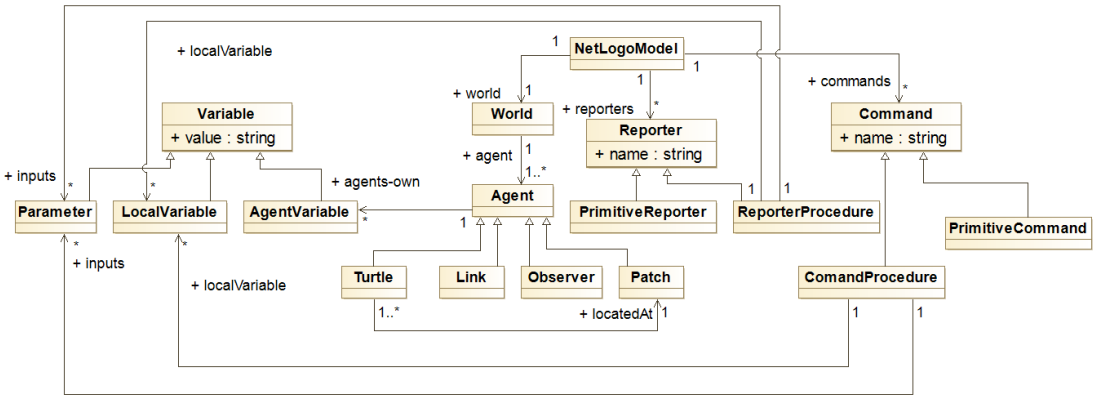


Figure 6.6 – B-reactive editor after implementation of validation rules

In NetLogo, commands and reporters tell agents what to do. A command is an action for an agent to carry out, resulting in some effect. A reporter is an instruction for computing a value, which the agent then "reports" to whoever asked it. Usually, a command name begins with a verb, such as `create`, `die`, `jump`, `inspect`, or `clear`". Most reporter names are nouns or noun phrases.

There are two types of procedures in Netlogo: those that are built into NetLogo (called primitives) and those defined by the user (called user defined). Commands and reports have a name. They are preceded by the keyword `to` or `to-report`, depending on whether it is a command procedure or a reporter procedure. The keyword `end` marks the end of the commands in the procedure. Once a procedure is defined, it can be used elsewhere in the model. Commands and reporters may take inputs: these are values that the command or reporter uses in carrying out its actions or computing its result.

Procedures can take inputs, just like many primitives do. To create a procedure that accepts inputs, put their names in square brackets after the procedure name. Netlogo has two essential command procedures (user-defined) that must be specified in the model: the `setup` and `go` procedures. The `go` procedure defines the order of what NetLogo will perform during simulation and thus, can be considered as NetLogo's schedulers procedure, where time passes in discrete steps called `tick`.

Agent variables are places to store values (such as numbers) in an agent. An agent variable can be a global variable, a turtle variable, a patch variable, or a link variable. If a variable is a global variable, there is only one value for this variable, and each agent can access it. Turtle, patch, and link variables are different: each turtle has its own value for every turtle variable and the same goes for patches and links.

Considering the presented general structure of Netlogo, the B-Reactive metamodel, and the Acceleo good practices of template organization, the following package struture is proposed for NetLogo code generator: a main template per generator (called generate), a template per file generated (called generateNetlogo) and a package (called common) containing many templates for behaviors (and its elements) transformations into NetLogo procedures. The main template and the file genetator template are implemented in appendix code C.1 and C.2 respectively. They include import declarations, output file definition, and they call NetLogo agent's template generators (turtles and patches), setup procedures generators, agent's procedures generators and the `go` procedure generator.

6.5.1.1 Generating Breedings, Turtles and Patches

In NetLogo, patches and turtles are defined with keyword "breed", and like globals, `turtles-own`, and `patches-own` keywords (used to define NetLogo's agents attributes), it can only be used at the beginning of the Code tab, before any procedure definitions. In appendix code C.3, a `breed` is defined for every B-reactive entity, where `turles-own` and `patches-own` attributes are also generated.

6.5.1.2 Generating Setup procedures

Turtles and patches are later initialized in `setup` procedures. But in order to create turtles and patches, one question remains: how to distinguish entities

that will become patches from the ones that will become turtles ? The answer relies one the way we initialize our agents with B-Reactive. In both models shown in appendix code A, if we initialize entities using Entity Set function call expressions, turtles will be created. If on the other hand, if a space of entities made of agents is declared, then a NetLogo grid of patches is created. The whole `setup` procedure for turtles is divided into appendix codes C.4 and C.5. First, the turtles are transformed from B-reactive's entities, their initial position is defined in and all their variables (parameters and attributes) are initialized. A special report procedure is created for every turtle in the model, called `one-of entity's name-here`. This is done to facilitate later turtle's reference

Next, the code generator create a set of turtles that simulates the environment. However, this approach creates a set of turtles, placed in the environment and for that reason, it is not possible to inspect (observe) Netlogo's patches. Since spatial entities possess some methods that are different from agents' methods (like many MAS), predicting the correct scope for each of those procedures may highly increase the complexity of code generation. By considering the environment as a set of spatial entities, it facilitates code generation into Netlogo code because turtles will have the same type of procedures. NetLogo's `setup` procedure is realized for patches and turtles. The appendix code in C.5 shows how the space is initialized in NetLogo.

NetLogo's function `resize-world` uses values defined in B-reactive's entity initialization to create a grid (environment) of x size per y cells. The code performs a loop to create the correct number of cells in the space and put static entities on each cell. Next, a turtle per cell is created, and the environment is initializes. Turtles are set to `hidden` to better visualize other turtles.

6.5.1.3 Generating command and reporter procedures

To generate procedures in NetLogo, we created a main template (appendix code C.6) that calls other templates to individually treat for each type of behavior specified in B-reactive models. More precisely, B-reactive EquationBehaviors, ActivityBehaviors and ActivityDiagramBehaviors are detected by the appendix template C.6, that in turn, calls an specific template to deal with M2T transformation.

To transform B-reactive EquationBehaviors, the appendix code C.7 generates all possible local variables and equations. Since the values are initialized in the NetLogo's `setup` procedure, the parameters are not generated here. We used

the same strategy for all succeeding behaviors. Activity behaviors are transformed into NetLogo's procedures according to appendix template code C.8). This template declares all local variables and calls another the template (see appendix code C.9) that is responsible for mapping every B-reactive's primitive activity (such as `Add`, `Remove`, `Set`, etc) into Netlogo's code.

Last, in order to generate B-reactive's ActivityDiagramBehavior, every node and edge found are transformed in template code C.10. Activity diagram behaviors template starts by importing a template that defines transformation of nodes and edges. This template carries? the first Node (Start) transformation that calls other elements of an activity diagram behavior until all nodes and edges are transformed.

In addition, we still have a considerable number of templates that implement function call expressions that are necessary to specific procedures transformation in Netlogo. One example, is the arithmetic function call expression for equation generations. Other templates are implementations of recursive expressions that try to detect the correct procedure to be translated into code. Therefore, for the sake of simplicity, the explanations of templates with exhaustive template calls were deliberately omitted from this chapter.

6.5.1.4 Generating the go procedure

This code should be self-explanatory. The `go` procedure is equivalent to B-reactive's scheduler, i.e. a modeler declares what (and in what sequence) procedures will be executed during model simulation. Since B-reactive does not have such a method, the behavior definition followed by the keyword `as main`, in B-reactive language will be transformed into Netlogo's `go` procedure, as highlighted in appendix code C.11

6.5.2 Code generation of Cormas methods

In Smalltalk (Cormas programing language), the message is the most basic construction. Control structures are nothing more than message passing : messages have receivers and selectors, but messages can also have arguments. For example, in the expression `2 raisedTo:4`, `2` is the receiver, `raisedTo` is the message, and `4` is the message argument. Messages can also be assigned to variables, like in `aVariable := 2 raisedTo:4`, resulting in 16.

Similarly, in `(x>1) ifTrue:[Transcriptshow:"bigger"]`, the block `ifTrue:[]` is a message to the boolean expression `(x>1)`. In that sense, we can say that every expression is a message sent. Messages can be unary (i.e. transcript, factorial, squared,etc.), binary (i.e. +,-, etc.) or keyword messages (i.e. raidsedTo, modulo, etc.). Each statement will have one or more smalltalk expression. A valid Smalltalk expression can be a variable name, a literal, a keyword, a statement, a block or a method.

Differently from object-orienrted langauges such as Java and C++, Smalltalk does not need constructors: instead, we can just create a class method that can take parameters. For instance, the method `newName:age:` of the class `Person` can be invoked by using the statement `p := Person newName: 'Ada Lovelace' age: 201 .`

In VisualWorks, a class has shared variables and namespace shared variables, while Squeak and many other implementations have class variables, pool variables and global variables. Temporary variable declarations in Smalltalk are variables declared inside a method. As in many SmallTalk environments, Cormas makes use of protocols. A protocol is a way to organize methods that ideally, indicate the methods' intent in the protocol's name. For that reason, some common protocol names have been established by convention. For example, the accessing protocol for all accessors methods, the init protocol for establishing a consistent initial state for the object, and the control protocol to send commands to the model and update its state.

A non exhaustive abstract syntax that represents the discussed concepts is depicted in figure 6.7

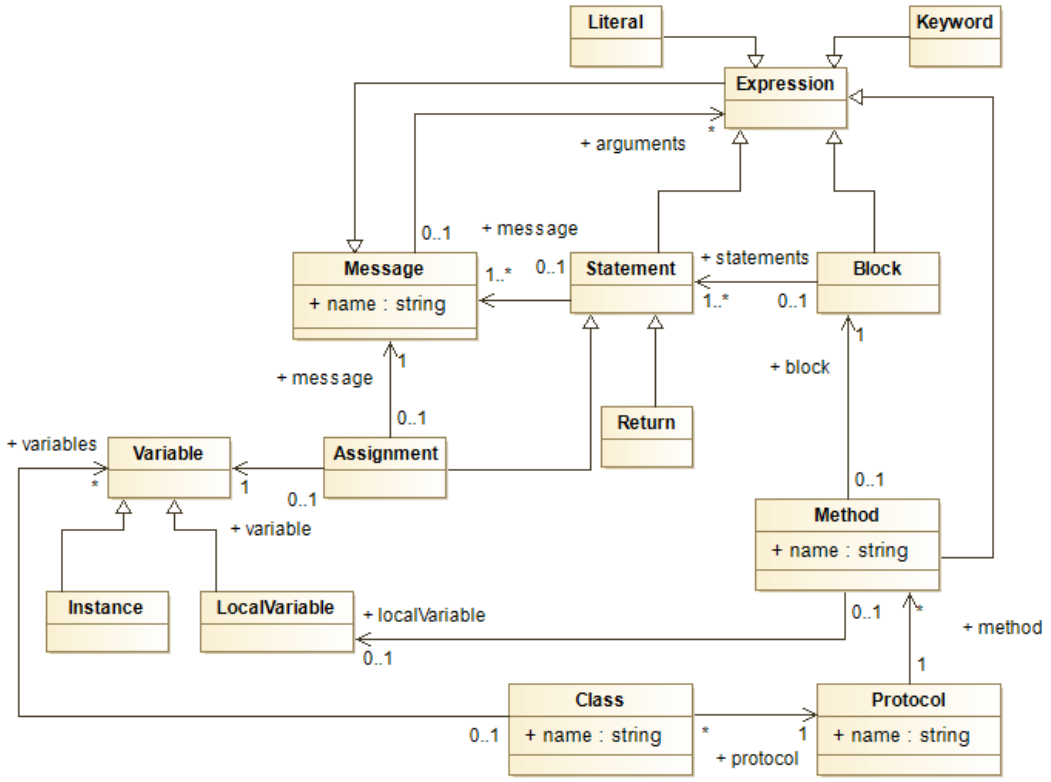


Figure 6.7 – B-reactive editor after implementation of validation rules

The templates for code generation are organized as follows: a template for generating Cormas core classes and template for generating model classes. The first generates main classes used by the kernel of Cormas, but also generates code for the accessing, init, control, instance-creation and probes protocols. The second template generate classes that for the defined entities in the model, as well as methods for their correspondent protocols. The generated code however, strictly follows the .pst visualworks file structure, although Cormas will be ported in future to Pharo - an open source SmallTalk environment development.

6.5.2.1 Generating Cormas classes

In object-oriented programing languages (like SmallTalk), there are two types of variables that a class may define. The first is the Class variables type: it occurs when just one copy of a class variable is shared with all instances of a class. This means class variables cannot be modified by the instances of that class. The second is the Instance variables type: it occurs when variables of a class also belong to the instances of that class. This means that these variables can have their value modified.

Although B-reactive does not explicitly implement such mechanism, we assumed that it is safe enough to transform entities' attributes of B-reactive language into instance variables, and parameters defined in Breactive's behaviors into class variables. The transformation into class variables and instance variables are implemented as shown in appendix code C.12. CormasModel classes, `AgentLocation` class, and `SpatialEntityCell` general structure are also generated. The template also declares some POVs (for point-of-view), a special type of attribute that is used to define points of views in Cormas. We will briefly discuss more Cormas POVs in further sections.

6.5.2.2 Generating methods for the accessing protocol

In Cormas, the accessing protocol is created to group two types of accessors methods that are very commonly used in object oriented programing: getters and setters. For the CormasModel class, Cormas automatically generates code for getters and setters for agents, for every attribute of B-reactive entities. Accessors methods are generated for all initialized parameters and attributes of B-reactive language. Next, a getter method is specified (for each Entity). The getter method returns a collection of all instances of aClass (and sub classes) collected by the CormasModel class. For every entity of B-reactive language, Cormas usually generated a getter and a setter method for a special attribute. This attribute name (also generated by default), is called `theAgentNames`. That attribute returns a list of cells (if it is a SpatialEntityCell) or a list of agents (if it is an `AgentLocation` class) and it is later used to define probes (see section for C.17). Finally, getter setter methods are also generated for every B-reactive at-tributes that define the initial number of agents and cells, according to B-reactive's initialization and instantiation. The accessing protocol is generated according to the appendix code, C.13,

6.5.2.3 Generating methods for the instance-creation protocol

The instance-creation protocol organizes methods that are responsible for defining how instances of `AgentLocation` and `SpatialEntityCell` classes are created. In the appendix code C.14 the environment is initialized using the the Cormas method `initializeRegular`. The method responsible for the environment initialization requires some parameters, such as the number of lines and columns, the shape of cells, the number of neighbors, and the shape of

boundaries. Lines and columns are provided by the InitSpace definition of B-reactive model. The other parameters are generated with default values : "eight" for the number of neighbors, `#squared` for the shape of cells, and `#toroidal` for the shape of boundaries. The appendix code also generated code to initialize the agents by calling the method `initAgents`.

The *initAgents* method must be generated for every subclass of `AgentLocation` class. Although B-reactive does not explicitly support the heritage mechanism, B-reactive offers the possibility to define an "alias" for every entity, during model initialization. If an "alias" is defined, the code is accordingly generated, simulating the heritage mechanism. If not (meaning that the entities do not have subclasses), the `initAgent` method is generated using the class name for variables definition.

6.5.2.4 Generating methods for init protocol

In general, the init protocol contains methods that initialize the environment in a simulation. In Cormas, the environment can be defined by the user, or it can be loaded as predefined grids that are manually defined. Since environments are usually defined by external files in Cormas, the appendix template code C.15 only generates code that references those predefined environments The first predefined environment is called `noAgents` and it is useful to run a simulation without any `AgentLocation`, just to show how `SpatialEntityCells` behaves if any behavior is associated to them. Note that the previously `initAgent` method (defined in the init-creation protocol) is not called here. The second predefined environment is called `homogeneousEnv` and initialized agents to be distributed in an homogeneous environment. The last (called fragmentedEnv) initializes a fragmented environment composed by patches of `SpatialEntityCell` units.

6.5.2.5 Generating methods for control protocol

The control protocol contains a sequence of all methods that are performed by every agent and spatial entity, according to the model's specification: these methods instantiate the agents and the environment and set the links between the entities. In this sense, the control protocol generation is very straightforward, since it defines all behaviors to be performed by every entity in the model. But the main class of Cormas (CormasModel) does not possesses any behaviors associated to it. Instead, the control protocol of the main class schedules entities

by sending a step to each one of these entities. Cormas is oriented towards step-by-step simulations (although it is possible to program discrete events simulations. The step method is generated as highlighted in code appendix code C.16. At each step, the environment is activated by pointing to previously generated probes (session C.17) and agents are randomly activated according their own definition for the step method.

6.5.2.6 Generating methods for probes protocol

Probes are special methods in Cormas used to record the variations of markers : they return changes occurred in any desired variable. By default, we define probes that return the population of every `AgentLocation` (B-reactive entity) and every `SpatialEntityCell` attribute, that is the attributes of the entities that represent the grid of location in B-reactive langauge. The probes code generation is shown in appendix code C.17.

6.5.2.7 Generating methods for custom protocols

We define custom protocols as a protocol containing all methods that defined by the user. By default, this protocol is called `GeneratedProtocol`. Every behavior that is defined the user in B-reative language is organized inse the `GeneratedProtocol`. Methods are transformed in almost the same manner as Netlogo: template Expressions evaluate the type of expressions, and, if a function is called, then for each function domain, another template is called until the expression is fully evaluated and transformed into code.

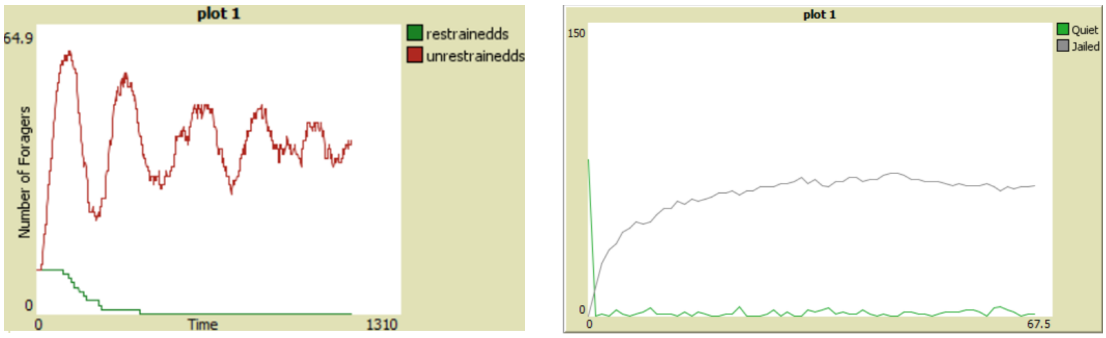
Cormas also generates many other protocols such as "-probes" (to define probes), "info", "description", "pov symbols", etc. Each class also defines its own accessing, init, control and many of the protocol discussed so far. Additionally, we implemented code generation for methods that are responsible for observing the entities (called POV). They are generated with a certain degree of automation, thanks to the use of Acceleo queries. Acceleo queries allow using java language, further increasing the possibilities of text generation. Appendix code C.18 shows the generation of custom protocols.

6.6 Model simulation

As a final objective of our approach, we aimed at performing direct simulation from the generated code. In order to achieve this, two models were implemented in B-reactive language: the ECEC model and the prison rebelion model. These models were implemented in B-reactive language, as shown in appendix A. With some visualization customization, we were able to simulate them in Netlogo. Although most of the methods were successfully generated in Cor-mas, the simulation could not be directly performed from the source code obtained, for reasons explained in section 6.6.2.

6.6.1 Netlogo simulation

Figure 6.8 shows results for the ECEC model (6.8a) and prison rebellion model (6.8b) simulations, respectively. In (6.8a) the simulation shows a very simple plot visualization: it computes the population of both forager species (Restrained and Unrestrained) over time. Initially, the unrestrained foragers will always prevail over the restrained foragers, since their feeding is not restricted. In a standard scenario we could say that this is an expected scenario for a first observation This could change, of course, depending on whether we also change certain initial parameters of the model, or on what we wish to observe.



(a) ECEC model simulation (b) Rebellion model simulation

Figure 6.8 – Model simulation from generated code in Netlogo

The prison rebellion also shows an expected behavior: the population of prisoners to be jailed decreases (in the same proportion) as the population of prisoners who are quiet increases. That means that law enforcements are being applied to prisoners that are considered subversive, leaving the quiet ones into an observation status by the agents jail. As in the ECEC model, we have chosen

to observe only the prisoners population size during the simulation, according to their situation (quiet or jailed).Observing other variables or setting different values for some parameters would certainly produce distinct plots for that model.

6.6.2 Cormas simulation

At the time of writing, the Cormas official version was still based on VisualWorks, a proprietary cross-platform implementation of Smalltalk language. It is implemented as a development system based on "images", which are dynamic collections of software objects, each contained in a system image. Visualworks source code (from .pst files) into parcel files (.pcl). that contains only compiled code in a binary format. However, Visualworks may also store source code in a number of different places. Additionally, there is no interface or command line (as far as we know) that allows VisualWorks VM to translate .pst (source code) to .plc (binary) files. For those reasons, we were not able to directly perform model simulation in Cormas from the generated code. Still, some codes could be successfully generated for Cormas (see code B.1 for ECEC model and code B.2 for the prison rebellion model)

6.7 Conclusion

In this chapter, we showed how MDE was applied to implement a proposed language to represent reactive behavior in multi-agent systems. Starting from a meta-model conceived with stakeholders, we implemented an Xtext editor and added validation rules and icons outline that highlight code to facilitate the modeling process. Next,we developed two code generators targeting Netlogo and Cormas platforms. For each code generator, different strategies were established considering the platform structure and the abstract syntax of each of the target MAS platforms In a cyclic process, as long as the generated code was tested and, if not executable, (or not valid), then new grammar and validation rules were added, re-starting the process. As a result, we could obtain executable code generation for Netlogo, and a reasonable code generation for Cormas. The generated codes were directly obtained from SES models implemented in B-Reactive models. Although code generation was not fully performed (due

to time constraints), we consider MDE as a fully capable approach for providing tools and quick customization options to implement solutions that solve the issues expounded so far.

Chapter 7

General conclusion

7.1 Discussion

7.1.1 MDE as an approach for designing DSL for SES

In this thesis, we developed a domain-specific language to facilitate the specification of reactive behaviors of MAS. We aimed to overcome three of the main issues found in the domain of SES during the specification of behaviors in MAS.

The first issue is the lack of an accessible language to specify the MAS system's dynamics (in that case, the behavior of an agent). Most MAS developers seldom took into consideration what are the terms used by non-programmers and domains-experts to describe a behavior. To overcome this , we used a meta-model conception approach, based on the discourse of SES domain-experts. This discourse was captured by analyzing the semantic structure of a SES model described in a natural or more formal language. Once understood, that discourse was used to increase the abstraction level of a meta-model that we conceptualized

Much of the discourse of SES domain-experts relies on informal definitions about certain modeling concepts. To overcome this, a modeling language (called B-reactive) was developed to provide domain-experts with an easier way to express behaviors in MAS. Based on a previous SES, we were able to provide a detailed specification of the three conceptual levels that are usually involved in the implementation of a modeling language: the abstract syntax (to describe what is the dis-course of domain experts), the concrete syntax (to describe how domain experts express their discourse) and the semantic level (to describe what the discourse employed by domain experts is about).

From the perspective of choosing MDE to develop an abstract syntax, we chose the meta-model centered approach. This approach increases the level of abstraction of a DSL, but also it is certainly the slowest one. Despite the fact that the ECore is very similar to UML and that the EMF provides a good set of visual for modeling an Ecore meta-model, Ecore is just a fraction of the full UML specification. For that reason, if the complexity level of the DSL increases and a meta-model approach is chosen, some adaptations to the final Ecore meta-model will eventually be required. These adaptations include some modifications in the EMF generator and the addition of some constraints to the Ecore meta-model that can only be achieved by learning EMF and OCL.

From the perspective of choosing MDE to develop the concrete syntax, our initial aim was to develop a DSL editor that would be as close as possible to a natural language. But some grammar rules specified in Xtext had to be very of-ten changed to fit its modified ANTRL parser. As all EMF frameworks, Xtext is strongly based on EMF and the grammar should respect some of EMF philosophy. Although EMF is easy to learn, most MDE frameworks require a deep knowledge of the internals of the EMP and a lot of programing. That was one of the major constraints of this work, specially in its early stages, when at the same time the meta-model was being conceptualized and EMF learning was required.

The second issue is that most programming languages used by MAS plat-forms to describe the behavior of agents are platform-dependent. This means that a model specified in one MAS platform cannot be properly used in a second MAS platform due to the incompatibility of programming languages or tools' specificities. Hence, a common layer between MAS platforms is necessary. At the same time, this layer should be easily accessible to domain-experts with virtually no programming language skills. M2T transformation plays an essential role in building up the interoperability layer between MAS since code-generation can effectively decrease MAS platform dependence. Nonetheless, code generation efficiency relies on a deep knowledge of target platforms, specially when platforms (MAS) specificities and programming languages should be taken into consideration. For that reason, it is essential to develop a meta-model with a high-enough level of abstraction to reach this purpose.

The last issue is the lack of a graphical formalism to represent reactive behaviors in multi-agent systems. MDE provides a good set of tools in compliance with most OMG's standards that are frequently used in the domain of software

engineering. There are many tools (graphical and textual) that could be potentially used to translate (or help to formalize) some informal terms used by domain-experts of SES. Additionally, the developed editor is still built as an eclipse plug-in. If an RCP (Rich-Client Application) is aimed to be a final DSL product, dealing with the powerful, but vast EMF is inevitable.

7.1.2 Cyclic approach for developing a DSL

The meta-model provided a sufficient level of abstraction to target other MAS platforms but, as stated, it significantly reduces the speed of a DSL implementation. One way to improve this could be the development of quick prototypes of the concrete syntax, with continuous testing performed with domain experts. Although the inferred meta-model would be significantly different from the one we conceived, time constraints did not allow to evaluate whether, in fact, a gram-mar centric approach could speed up the DSL development process. It is certain, however, that had prototypes been provided to domain-experts in the early stages of the DSL, this would have increased DSL feedback , and consequently would have increased the speed of the DSL implementation, regardless of the chosen approach (grammar or meta-model centred)

Transformations are not always perfect. Best practices are often context dependent - what is optimal in one context may be suboptimal in another. Acceleo compensates the little freedom in language design by adding rich tooling. In that sense, perhaps it would be more efficient to perform a model-to-model transformation before performing model-to-text transformation. More precisely, a desirable scenario would be the porting the meta-models of procedures and methods into Ecore, perform a M2M transformation from B-reactive model to Netlogo or Cormas meta-models, and later, apply a M2T transformation.

7.1.3 Evaluation of DSL and simulation of generated code

Model-driven engineering can indeed reduce, but not remove the inherent complexity that exists in software development; Although the proposed language could provide enough expressibility to implement two SES, there is still much to be done for improving the syntax of B-reactive language, specially regarding the set of predefined functions or a mechanism to extend the language. In Cormas for instance, if we wish to create the location that represents the nearest spatial entity within a given radius and with no occupant of any kind, we just have to

use the method `nearestEmptyLocationWithinRadius`. Because B-reactive language has no equivalent function, the code is manually generated to simulate the behavior of function `nearestEmptyLocationWithinRadius`. The same happens between Cormas method `randomWalkConstrainedBy` and B-reactive's function `selectConditionedLocation`. By consequence, the use of local variables is very often required to reduce expressions into smaller parts, decreasing the complexity of code generators implementation.

Another aspect is the distinction between the environment and the agents. Although these terms are specific to MAS (not SES), the meta-model provided enough abstraction to separate entities that represent the environment from entities that represent agents. However, the majority of MAS tools apply very specific functions to agents and to environments. This brings complexity to code generation since the way that code is generated may drastically change depending on what type of entity is represented in the model. A possible solution could be to add a distinct representation for `Entity` and `Environment` that would allow a direct mapping between B-Reactive's `Entity` into Cormas' `AgentLocation` and Netlogo's `turtle`. In the same sense, B-Reactive `Environment` could be easily mapped to Cormas `SpaceEntityCell` and Netlogo's `patch`.

7.2 Future works

While this thesis has demonstrated the potential of efficiently using MDE to implement a DSL for SES to facilitate the modeling of reactive behavior in MAS, many opportunities for extending the scope of this thesis remain. This section presents some of these directions, which include:

- The improvement of the meta-model to support other types of equation and formalisms to describe a given behavior. Although non-programmers might not be familiar with some formalisms such as logistic regression and differential equation, researchers would likely appreciate the possibility to describe MAS behaviors using other types of equations. Moreover, new types of formalisms (such as DEVS, CSP, Z, etc) could also be included in the meta-model, thus increasing the flexibility of code generation to MAS platforms (such as MIMOSa) that are based on such formalisms. Finally, with new elements that cover most of the MAS platforms, the meta-model

could be refined to support code generation to a third MAS plat-form, such as GAMA.

- The addition of other validation rules can help to ensure the validity of a model and could prevent the generation of erroneous codes. With the specification of more complex behaviors, validation rules may provide some semantic checks even if the compiler does not find any compile-time errors. By using the quick fix mechanisms provided by Xtend, the user can easily make a suggestion to fix an error, reducing the possibility of syntax and semantic errors.

- The implementation of a graphic editor to represent some elements of activity diagrams to improve the modeling process. With a graphical editor, behaviors expressed as activity behaviors or activity diagram behaviors would be easily expressed by non-programmers. However, certain aspects (such as equations) are not as efficiently represented with graphical icons. But this could be easily overcome through the development of editors combining both graphical and textual representation of behaviors.

- Model point-of-view capabilities should be incorporate-rated to B-Reactive. Although we provide a very limited code-generation of point-of-views during Cormas code generation, B-Reactive was not initially conceived to describe MAS point-of-views. Point-of-views are very useful and are an essential aspect of MAS M&S because they allow a user to choose what he wishes to observe during simulation (e.g. the value of an attribute or the population of an agent over time). Ideally, a new layer (abstract syntax) for point-of-view specification should be designed along with a concrete syntax to be incorporated to B-Reactive language.

- The realization of tests of our proposed DSL during participatory modeling workshops. Due to time limitations, we were not able to effectively test our language in a real-case scenario. Even though the B-reactive language is still a prototype, B-Reactive can be improved trough the feedback of domain experts. This feedback can be captured in such events as training sessions and workshops, where the needs of domain experts during model and simulation stages are more easily identified.

7.3 Conclusion

B-reactive is a prototype of DSL to model reactive behaviors on MAS. Built with MDE tools, the language aims at enhancing the level of participation of SES domain-experts with no programming skills. Additionally, the language was conceived by applying a meta-model approach that aimed to decrease MAS models' dependence to different platforms. The meta-model served as an interoperable layer from which model-to-text transformations can be easily applied to generate code for various MAS platforms. Meanwhile, B-Reactive represents the intent to look at a different direction for establishing a modeling environment (starting with a language) that is truly suited to non-programmers, while , providing them at the same time the means to apply their own discourse during MAS modeling and simulation. While it is true that there is no silver bullet to overcome MAS model's complexity, there is much theoretical work to be done tot address some of the complexities of SES and directly influences the way DSL are developed. One of these particularities is granularity. This requires the introduction of a new representation of formalisms and inference mechanisms such as aggregation and disaggregation. Hence, a large field of research still remains open to SES domain formalization. This work endeavours to be the first step toward a further understanding of these important issues.

Appendix A

SES axmodels in B-Reactive language

A.1 Implementation of ECEC model in B-Reactive language

```
Model Ecec {

 Entity Plant  {
  Attributes { biomass: Float }
  EquationBehaviour grow {
   Parameters (k : Float,r : Float)
   Equation {biomass = biomass + r*biomass * (1-biomass/k)}
  }
 }

 Entity Forager {
  Attributes {energy: Float}
  ActivityBehavior ConsumeEnergy {
   Parameters (catabolic_rate:Float)
   Remove catabolic_rate from energy
  }

  ActivityDiagramBehavior ToMove  {
   let biomassOfPlant <- Plant.biomass of Plant from here
   let aLocation <- max-one-of [ Plant.biomass,
   select-location-from [ neighborhood ] such that (
   neighborhood is NOT occupied by (any Forager here) )
   union-location (here) ]
```

```
21    Start -> Decide { if (biomassOfPlant >= ConsumeEnergy.
      catabolic_rate) then Move to {aLocation} -> End else Move to
       {one-of[ union-location(neighborhood,here) ]} -> End }
22   }
23
24   ActivityBehavior Eat  {
25    Parameters ( harvest_rate: Float )
26    let aPlantBiomass <- Plant.biomass of one-of (Plant from
      here )
27    Add harvest_rate * aPlantBiomass to Plant.biomass
28    Remove harvest_rate * aPlantBiomass from Plant.biomass
29   }
30
31   ActivityDiagramBehavior ToReproduce {
32    Parameters ( Fertility_Threshold: Float )
33    Start -> Decide { if ( energy >= Fertility_Threshold ) then
      Reproduce(1) with energy (50) placed on one-of [neighborhood
      ] -> Remove 50 from energy -> End
34    }
35   }
36
37   ActivityDiagramBehavior ToDie  {
38    Start -> Decide  { if  ( energy < 0 )  then Die -> End }
39   }
40  }
41
42  Run main as : ActivityDiagramBehavior Main {
43   Start -> Plant.grow ->Forager.ConsumeEnergy -> Forager.Eat ->
       Forager.ToMove -> Forager.ToReproduce -> Forager.ToDie ->
      End
44  }
45  //-------------------------------------------|
46  //             INITIALIZATION       |
47  //-------------------------------------------|
48  //------------Space init--------------------
49  Create Forager 10 as Restrained{
50   each Forager {
51    position = one-of [ grid of Plant ]
52    Forager.ConsumeEnergy.catabolic_rate := 2
53    Forager.Eat.harvest_rate := 0.5
54    Forager.energy := 50
```

```
    Forager.ToReproduce.Fertility_Threshold := 100
   }
  }

  Create  Forager 10 as Unrestrained {
   each Forager {
    position = one-of [ grid of Plant ]
    Forager.ConsumeEnergy.catabolic_rate := 2
    Forager.Eat.harvest_rate := 0.9
    Forager.energy := 50
    Forager.ToReproduce.Fertility_Threshold := 100
   }
  }
  //----------Entity init-----------------
  Create grid of Plant (20,20) {
   each Plant {
    Plant.grow.k := 10
    Plant.grow.r := 0.2
    Plant.biomass := random-float (Plant.grow.k)
   }
  }
}
```

A.2 Implementation of prison rebellion model B-Reactive language

```
Model AgentsAndCops {

 Entity agent {
  Attributes {
   active :  Boolean ,
   movement:Boolean,
   jailterm : Int,
   arrestProbability : Float,grievance : Float,
   riskAversion:Float,
   perceivedhardship : Float,
   governmentlegitimacy:Float
  }

  EquationBehaviour estimateArrestProbability {
   Parameters (k : Float )
   let c <- count ( cop  neighborhood )
   let a <- count  ( agent neighborhood having  ( get active
   true ) )
   Equation {
    arrestProbability = 1 - ( exp ( - k * ( floor ( c / (a+1) )
    ) ) )
   }
  }

  ActivityDiagramBehavior DetermineBehavior {
   let test <- grievance - riskAversion * arrestProbability
   Start -> Decide  { if ( test > cop.enforce.threshold )  then
    Set active := true else
    Set active := false -> End
   }
  }

  ActivityDiagramBehavior MoveAgent {
   let targets  <- select a location from [ neighborhood ]such
   that ( neighborhood is NOT occupied  by ( any cop  here  ) )
    AND ( all agent on here has jailterm > 0 )
```

```
33    let numtargets <- count ( targets  )
34    Start -> Decide  { if  ( numtargets >= 1) then
35     Move to  { one-of [ targets ] } -> End }
36   }
37
38   ActivityDiagramBehavior ReduceJailTerm {
39    Start -> Decide  { if  ( jailterm > 1 ) then
40     Remove 1 from  jailterm  -> End }
41   }
42
43  }
44
45  Entity cop {
46
47   ActivityDiagramBehavior enforce {
48    Parameters ( maxJailTerm : Float,threshold : Float  )
49    let suspect <- one-of (agent from  neighborhood having ((get
       agent.active true )))
50    let numsuspect  <- count (suspect)
51    Start -> Decide  { if  ( numsuspect > 0 )  then
52     Set agent.active := false -> Set agent.jailterm :=
      random-int ( maxJailTerm ) -> Move to { one-of [suspect] }
      -> End
53    }
54   }
55
56   ActivityDiagramBehavior MoveCops {
57    let targets  <- select a location from [ neighborhood ]such
      that (neighborhood is NOT occupied  by ( any cop  here  ) )
      AND( all agent on here has agent.jailterm > 0)
58    let numtargets <- count ( targets  )
59    Start -> Decide  { if  ( numtargets >= 1)  then
60     Move to  { one-of [ targets ] } -> End }
61   }
62  }
63
64  Entity cell {}
65
66  Run main as :  ActivityDiagramBehavior MainAgentsAndCops {
67
```

```
68    Start -> agent.estimateArrestProbability  -> agent.MoveAgent
       -> agent.DetermineBehavior -> agent.ReduceJailTerm -> cop.
       MoveCops -> cop.enforce -> End
69   }
70   Create agent 20 as prisonerAgent {
71    each agent {
72     position = one-of [ select a location from [ grid ]
73     such that (grid is NOT occupied by ( any agent here , any
       cop here )) ]
74     agent.riskAversion := 1.0
75     agent.perceivedhardship := 1.0
76     agent.jailterm := 0
77     agent.governmentlegitimacy := 0.83
78     agent.active := false
79     agent.estimateArrestProbability.k := 2.3
80     cop.enforce.threshold := 0.1
81     agent.grievance := agent.perceivedhardship * 1 - agent.
       governmentlegitimacy
82    }
83   }
84   Create cop 20 as copAgent {
85    each cop {
86     position = one-of [select a location from [ grid ]such that
87     (grid is NOT occupied by ( any agent here , any cop here ) )
        ]
88     cop.enforce.maxJailTerm := 30
89    }
90   }
91   Create grid of cell (20,20) {
92    each cell { }
93   }
94  }
```

Appendix B

Generated code

B.1 Cormas generated code for ECEC model

Code B.1 – Generated instance-creation method of ECEC model

```
1  "The init enviromment method"
2  initEnviromment
3          self spaceModel initializeRegularX: 20.0 Y: 20.0 shape: #squared nbNeighbours:
               #eight boundaries: #toroidal
4          self thePlants do:[: cell | cell initRandomBiomass].
5          self initAgents
6
7  "The init agents method"
8  initAgents
9          self createN: self restrainedInitialNumber randomlyLocatedAloneEntities:
               Restrained
10         self createN: self unrestrainedInitialNumber randomlyLocatedAloneEntities:
               Unrestrained
11
12 "The toEat agents method"
13 toEat
14 |aPlantBiomass |
15         aPlantBiomass := self patch neighboursMaxOf biomass
16         self energy: self energy + self harvest_rate * aplantbiomass
17         self biomass: self biomass - harvest_rate *aplantbiomass
18
19 "The ToMove method"
20 toMove
21 |aLocalVariable biomassOfPlant aLocation |
22         aLocation := self patch neighbourhoodAndSelf select : [cell: | cell biomass &gt;
               self class catabolicRate and :[ cell noOccupant]]
23         biomassOfPlant := self patch neighboursMaxOf biomass
24         self biomassofplant >= catabolicRate
25         ifTrue : [ self moveTo: alocation]
```

```
26          ifFalse : [self moveTo: (biomassOfPlant asSortedCollection:[:c1 :c2 |biomass &gt
                ; c2 biomass]) first]
27
28  "The ToReproduce method"
29          self energy >= fertility_threshold
30          ifTrue : [newborn:= self createN: 1 entity : Forager initMethod: #initEnergy.
31                    newborn randomWalkConstrainedBy [:c | c noOccupant]
32                    self energy:= energy - 50]
33
34  "The ToDie method"
35  ToDie
36          self energy < 0.0
37          ifTrue : [self dead: true]
38
39  "The toGrow method"
40  toGrow
41          self biomass biomass + r * biomass * 1.0 - biomass /k
```

B.2 Cormas generated code for Prison rebellion model

Code B.2 – Generated code for die behavior of ECEC model

```
1   "The estimateArrestProbability method"
2   estimateArrestProbability
3   |c a aPerceived|
4           c:= size perceivedEntities: Cop withinRange: 1.
5           aPerceived:= size perceivedEntities: Agent withinRange: 1
6           a:= size select: [:aPerceived | aPerceived #active:true ]
7           self arrestprobability 1.0 - (-k *(c /a +1.0 ) floor ) exp
8
9   "The DetermineBehavior method"
10  determineBehavior
11  |test riskaversion arrestprobability |
12          test:= grievance - riskaversion * arrestprobability
13          self test > threshold
14                    ifTrue : [self active:true]
15                    ilFalse :[self active:false]
16
17  "The enforce method"
18  enforce
19  |suspect numsuspect|
20          suspect perceivedEntities: agent withinRange: 1 select :[: active | active:= true]
```

```
21          numsuspect := size self suspect
22          self numsuspect > 0.0
23          ifTrue : [self active false.
24                  self jailterm := Cormas random < self maxjailterm.
25                  moveTo: Cormas selectRandomlyFrom: self suspect.]
```

B.3 Netlogo generated code for ECEC model

Code B.3 – Netlogo generated code for ECEC model

```
1  breed[plants plant]
2  breed[foragers forager]
3  foragers-own [ fertility_threshold  catabolic_rate  harvest_rate energy ]
4  plants-own [k r biomass ]
5  to setup
6  clear-all
7  setup-plants
8  setup-foragers
9  reset-ticks
10  ;; TODO should be implemented
11  end
12  to setup-foragers
13  ;; Start of user code Forager
14  ;; TODO should be implemented
15  ;; End of user code
16  create-foragers (10.0) [
17  move-to one-of patches
18  set catabolic_rate 2.0
19  set harvest_rate 0.5
20  set energy 50.0
21  set  fertility_threshold  100.0
22  ]
23  create-foragers (10.0) [
24  move-to one-of patches
25  set catabolic_rate 2.0
26  set harvest_rate 0.9
27  set energy 50.0
28  set  fertility_threshold  100.0
29  ]
30  end
31  to-report plant-here
32  report one-of plants-here
33  end
34  to setup-plants
```

```
35 resize-world 0 20.0 0 20.0
36 let i 0
37 let j 0
38 repeat 21 [
39 set j 0
40 repeat 21 [
41 create-plants 1[
42 setxy i j
43 set biomass random 1.0
44 set k 10.0
45 set r 0.2
46 set hidden? true
47 ]
48 set j j + 1
49 ]
50 set i i + 1
51 ]
52 end
53 to grow
54 set biomass biomass + r * k + 1.0 - biomass / k
55 end
56 to ConsumeEnergy
57 set energy energy - catabolic_rate
58 end
59 to Eat
60 let aPlantBiomass [biomass] of one-of plant-here
61 set energy energy - harvest_rate * aPlantBiomass
62 set biomass biomass - harvest_rate * aPlantBiomass
63 end
64 to ToMove
65 let aLocalVariable 5.0
66 let biomassOfPlant [biomass] of plant-here
67 let aLocation max-one-of (patch-set (patch-set patch-here ) neighbors with [ not any
       ? foragers])[[biomass] of plant-here]
68 ifelse biomassOfPlant >= catabolic_rate
69 [move-to aLocation]
70 [move-to one-of neighbors with [ not any? foragers]]
71 end
72 to ToReproduce
73 if Energy >= Fertility_Threshold
74 [hatch-foragers 1.0[ move-to one-of neighbors set energy(50.0) ]]
75 end
76 to ToDie
77 if Energy < 0.0
78 [die]
79 end
```

```
80 to step
81 ConsumeEnergy
82 Eat
83 ToMove
84 ToReproduce
85 ToDie
86 end
87 to go
88 ask turtles [
89 ;; TODO should add activities
90 ]
91 tick
92 end
```

B.4 Netlogo generated code for Prison Rebellion model

Code B.4 – Netlogo generated code for ECEC model

```
1 breed[agents agent]
2 breed[cops cop]
3 breed[cells cell ]
4 agents-own [k arrestprobability jailterm active riskaversion grievance
      perceivedhardship governmentlegitimacy]
5 cops-own [maxjailterm threshold]
6 cells-own [surroundinglocation ]
7 to setup
8 clear-all
9 setup-agents
10 setup-cops
11 setup-cells
12 reset-ticks
13 ;; TODO should be implemented
14 end
15 to setup-cops
16 ;; Start of user code cop
17 ;; TODO should be implemented
18 ;; End of user code
19 create-cops (64) [
20 move-to one-of patches with [not any? agents-here] with [ not any? cops-here ]
21 set threshold 0.1
22 set maxjailterm 30
23 set shape "circle"
```

```
24  set color blue
25  ]
26  end
27  to setup-agents
28   ;; Start of user code agent
29   ;; TODO should be implemented
30   ;; End of user code
31  create-agents (80) [
32  move-to one-of patches with [ not any? agents-here] with [ not any? cops-here ]
33  set riskaversion 1.0
34  set perceivedhardship 1.0
35  set governmentlegitimacy 0.83
36  set jailterm 0.0
37  set active false
38  set k 2.3
39  set grievance perceivedhardship * (1 - governmentlegitimacy)
40  ]
41  end
42  to-report cell-here
43  report one-of cells-here
44  end
45  to-report agent-here
46  report one-of agents-here
47  end
48  to-report cop-here
49  report one-of cops-here
50  end
51  to setup-cells
52  resize-world 0 40 0 40
53   let i 0
54   let j 0
55  repeat 41 [
56  set j 0
57  repeat 41 [
58  create-cells 1[
59  setxy i j
60  set hidden? true
61  ]
62  set j j + 1
63  ]
64  set i i + 1
65  ]
66  end
67  to estimatearrestprobability
68   let c count cops-on neighbors
69   let a count agents-here with [active]
```

```
70  set arrestprobability 1 - exp(- k * floor (c / (a + 1.0)))
71  end
72  to determinebehavior
73  let test grievance - riskaversion * arrestprobability
74  ask cops [
75  ifelse test > threshold
76  [ask agents [ set active true ] ]
77  [ask agents [ set active false ] ]
78  ]
79  ; print active
80  end
81  to moveagent
82  let targets neighbors with [ not any? cops-here and all? agents-here [jailterm > 0]]
83  let numtargets count targets
84  if numtargets >= 1
85  [move-to one-of targets]
86  end
87  to reducejailterm
88  if jailterm > 1.0
89  [set jailterm jailterm - 1.0]
90  end
91  ;; COPS BEHAVIOR
92  to enforce
93  let activeagents (agents-on neighbors) with [active]
94  let numactive count activeagents
95  if numactive >= 1 [
96  ;; arrest suspect
97  let suspect one-of activeagents
98  ask suspect [
99  set active false
100 set jailterm random [maxjailterm] of myself
101 ]
102 move-to suspect ;; move to patch of the jailed agent
103 ]
104 end
105 to movecops
106 let targets neighbors with [ not any? cops-here and all? agents-here [jailterm > 0]]
107 let numtargets count targets
108 if numtargets >= 1
109 [move-to one-of targets]
110 end
111 to go
112 ask patches [
113 ask cells []
114 ]
115 ask agents [
```

```
116  print  jailterm
117  estimatearrestprobability
118  ifelse  jailterm  = 0
119  [  determinebehavior moveagent ]
120  [  reducejailterm]
121  ]
122  ask cops [
123  movecops
124  enforce
125  ]
126  tick
127  end
```

Appendix C

M2T Acceleo templates

C.1 Netlogo M2T templates

Template C.1 – Acceleo file generator template

```
1  [comment encoding = UTF-8 /]
2  [module generate('/org.cirad.dsl.behavior.metamodel/model/
       metamodel.Ecore') /]
3  [import org::cirad::dsl::behavior::gen::netlogo::files::
       generateNetLogoFile /]
4  [template public generate(m : Model)]
5  [comment @main/]
6  [generateNetLogoFile(m)/]
7  [/template]
```

Template C.2 – Acceleo file generator template for Netlogo code generation

```
1  [comment encoding = UTF-8 /]
2  [module generateNetLogoFile('/org.cirad.dsl.behavior.
       metamodel/model/metamodel.Ecore') /]
3  [import org::cirad::dsl::behavior::gen::netlogo::commmon::
       generateTurtlesAndBreed/]
4  [import org::cirad::dsl::behavior::gen::netlogo::commmon::
       generateSetup/]
5  [import org::cirad::dsl::behavior::gen::netlogo::commmon::
       generateToGo/]
6  [import org::cirad::dsl::behavior::gen::netlogo::commmon::
       generateBehaviors/]
7  [template public generateNetLogoFile(m : Model)]
8  [file('generated_'.concat(m.name).concat('.nlogo'), false, '
       UTF-8')]
9  [generateBreed(m)/]
10 [generateTurtlesOwn(m)/] [generatePatchesOwn(m)/]
11 [generateSetup(m)/]
12 [generateBehaviors(m)/]
```

```
13      [generateToGo(m)/]
14      [/file]
15      [/template]
```

Template C.3 – Breed, patch and turtles declaration

```
1       [comment encoding = UTF-8 /]
2       [module generateTurtlesAndBreed('/org.cirad.dsl.behavior.
           metamodel/model/metamodel.Ecore')]
3       [import org::cirad::dsl::behavior::gen::netlogo::commmon::
           generateExpressions]
4       [template public generateBreed(aModel : Model)]
5       [for (anEntity:Entity | aModel.entities)]
6       breed['['/][anEntity.name.toLowerCase()/]s [anEntity.name.
           toLowerCase()/][']'/]
7       [/for]
8       [/template]
9       [template public generateTurtlesOwn(aModel : Model) ]
10      [for (initEnt : Entity | aModel.eAllContents(InitEntity).
           entity->asSet())]
11      [initEnt.name.toLower()/]s-own ['[' /][initEnt.behavior.
           parameters.generateArithimeticExpressions().toLower()->
           asSet().concat(' ')/][initEnt.attributes.
           generateArithimeticExpressions().toLower()->asSet().
           concat('')/][ ']' /]
12      [/for]
13      [/template]
14      [template public generatePatchesOwn(aModel : Model) ]
15      [for (initSpaceEntity: Entity | aModel.eAllContents(
           InitSpace).entity->asSet())]
16      [initSpaceEntity.name.toLower()/]s-own ['[' /][
           initSpaceEntity.behavior.parameters.
           generateArithimeticExpressions().toLower()->asSet().
           concat(' ')/][initSpaceEntity.attributes.
           generateArithimeticExpressions().toLower()->asSet().
           concat(' ')/][ ']' /]
17      [/for]
18      [/template]
```

Template C.4 – Code generation for turtles setup

```
1       [template public generateSetup(aModel : Model)]
2       to setup
3       clear-all
4       [for (anEntity:Entity | aModel.entities)]
5       setup-[anEntity.name.toLower()/]s
6       [/for]
```

```
7       reset-ticks
8       ;;TODO should be implemented
9       end
10      [for (anInitEntity:InitEntity | aModel.eAllContents(
            InitEntity)->asSet()) separator ('\n')]
11      to setup-[anInitEntity.entity.name.toLowerCase()/]s
12      ;;[protected (anInitEntity.entity.name)]
13      ;;TODO should be implemented
14      ;;[/protected]
15      create-[anInitEntity.entity.name.toLower()/]s ([aModel.
            eContents(InitEntity).initFunctionCall.oclAsType(
            FunctionCallExpression).arguments->last().
            generateArithimeticExpressions()/]) [ '[' /]
16      move-to [aModel.eContents(InitEntity)->first().
            initialLocation.generateLocationFunctionCallExpression()
            ->asSet()/]
17      [for (anAssignement:Assignment | anInitEntity.eAllContents(
            Assignment))]
18      set [anAssignement.variable.generateArithimeticExpressions()
            .toString().toLower()/] [anAssignement.expression.
            generateFunctionCallExpressions()/]
19      [/for]
20      [ ']' /]
21      end
22      [/for]
23      [for (entities : Entity | aModel.eAllContents(Entity))]
24      to-report [entities.name.toLower()/]-here
25      report one-of [entities.name.toLower()/]s-here
26      end
27      [/for]
```

Template C.5 – Code generation for environment setup

```
28      [for (anInitSpace:InitSpace|aModel.eAllContents(InitSpace))]
29      to setup-[anInitSpace.entity.name.toLower()/]s
30      [if (anInitSpace.initFunctionCall.eContents(NamedFunction).
            name->first() = 'Create grid')]
31      resize-world 0 [anInitSpace.initFunctionCall.oclAsType(
            FunctionCallExpression).arguments->at(2).
            generateArithimeticExpressions()/] 0 [anInitSpace.
            initFunctionCall.oclAsType(FunctionCallExpression).
            arguments->at(3).generateArithimeticExpressions()/]
32      let i 0
33      let j 0
```

```
34      repeat [anInitSpace.initFunctionCall.oclAsType(
           FunctionCallExpression).arguments->at(2).
           generateArithimeticExpressions().toReal().round() + 1 /]
           ['['/]
35      set j 0
36      repeat [anInitSpace.initFunctionCall.oclAsType(
           FunctionCallExpression).arguments->at(2).
           generateArithimeticExpressions().toReal().round() + 1/] [
           '['/]
37      create-[anInitSpace.entity.name.toLower()/]s 1['['/]
38      setxy i j
39      [for (anAssignement:Assignment | anInitSpace.eAllContents(
           Assignment))]
40      set [anAssignement.variable.generateArithimeticExpressions()
           /] [anAssignement.expression.
           generateFunctionCallExpressions()/]
41      [/for]
42      set hidden? true
43      [']'/]
44      set j j + 1
45      [']'/]
46      set i i + 1
47      [']'/]
48      [/if]
49      end
50      [/for]
51      [/template]
```

Template C.6 – Behavior

```
1       [module generateBehaviors('/org.cirad.dsl.behavior.metamodel
           /model/metamodel.Ecore')]
2       [import org::cirad::dsl::behavior::gen::netlogo::commmon::
           generateExpressions/]
3       [import org::cirad::dsl::behavior::gen::netlogo::commmon::
           generateEquations]
4       [import org::cirad::dsl::behavior::gen::netlogo::commmon::
           generateNodesAndEdges /]
5       [template public generateBehaviors(aModel : Model)]
6       [generateEquationBehaviors(aModel)/]
7       [generateActivityBehavior(aModel)/]
8       [generateActivityDiagramBehavior(aModel)/]
9       [/template]
```

Template C.7 – Transformation of Equation Behaviors into Netlogo procedures

```
1       [template public generateEquationBehaviors (aModel:Model)]
```

```
2       [for (anEquationBehavior : EquationBehavior | aModel.
           eAllContents(Entity).eAllContents(EquationBehavior))]
3       to [anEquationBehavior.name.toLower()/]
4       [for (aLocalVariable : LocalVariable | anEquationBehavior.
           eAllContents(LocalVariable)) separator('\n')]
5       let [aLocalVariable.generateArithimeticExpressions()/] [
           aLocalVariable.expression.generateFunctionCallExpressions
           ()/]
6       [/for]
7       [generateEquation(anEquationBehavior.equation)/]
8       end
9       [/for]
10      [/template]
```

Template C.8 – Transformation of Acvitity Behaviors into Netlogo procedures

```
1       [template public generateActivityBehavior (aModel:Model)]
2       [for (activityB:ActivityBehavior | aModel.eAllContents(
           Entity).eAllContents(ActivityBehavior))]
3       to [activityB.name.toLower()/]
4       [for (aLocalVariable : LocalVariable | activityB.
           eAllContents(LocalVariable)) separator('\n')]
5       let [aLocalVariable.generateArithimeticExpressions()/] [
           aLocalVariable.expression.generateFunctionCallExpressions
           ()/]
6       [/for]
7       [for (aPrimitiveActivity: PrimitiveActivities | activityB.
           eAllContents(PrimitiveActivities))] [aPrimitiveActivity.
           generatePrimitiveActivities()/]
8       [/for]
9       end
10      [/for]
11      [/template]
```

Template C.9 – Netlogo code generation into primitive activities

```
1       [template public generateAddNode(anAddNode : Add) post (
           replaceAll('\n', '').trim())]
2       set [anAddNode.toAttribute.generateArithimeticExpressions().
           toLower()/] [anAddNode.toAttribute.
           generateArithimeticExpressions().toString().toLower()/] -
            [anAddNode.expression.generateArithimeticExpressions().
           toLower()/]
3       [/template]
4       [template public generateRemoveNode(aRemoveNode : Remove)
           post (replaceAll('\n','').trim())]
```

```
5   set [aRemoveNode.from.generateArithimeticExpressions().
        toLower()/] [aRemoveNode.from.
        generateArithimeticExpressions().toLower()/] - [
        aRemoveNode.expression.generateArithimeticExpressions().
        toLower()/]
6   [/template]
7   [template public generateSetVariableNode(aSetNode :
        SetVariable)post (replaceAll('\n', '').trim())]
8   set [aSetNode.assignment.variable.
        generateArithimeticExpressions()/] [aSetNode.assignment.
        expression.generateFunctionCallExpressions()/]
9   [aSetNode.outcome_edge.generateEdges()/]
10  [/template]
11  [template public generateDieNode(aDieNode : Die)]
12  die
13  [/template]
14  [template public generateMoveNode(aMoveNode : Move)]
15  move-to [aMoveNode.locationexpression.
        generateFunctionCallExpressions()/]
16  [/template]
17  [template public generateReproduceNode(aReproduceNode :
        Reproduce) post (replaceAll('\n', '').trim())]
18  hatch-[aReproduceNode.ancestors(Entity).name.toLower()/]s [
        aReproduceNode.offspring_quantity.
        generateArithimeticExpressions()/]['['/] move-to [
        aReproduceNode.initial_location.
        generateFunctionCallExpressions()/]
19  set [aReproduceNode.declaredattributes.oclAsType(
        FunctionCallExpression).arguments->at(1).
        generateArithimeticExpressions().toLower()/]([
        aReproduceNode.declaredattributes.oclAsType(
        FunctionCallExpression).arguments->at(2).
        generateArithimeticExpressions().toLower()/]) [']'/]
20  [aReproduceNode.outcome_edge.generateEdges()/]
21  [/template]
```

Template C.10 – Mapping Acvitity Diagram Behaviors into Netlogo procedures

```
1   [template public generateActivityDiagramBehavior (aModel:
        Model) ]
2   [for (activityDB:ActivityDiagramBehavior | aModel.
        eAllContents(Entity).eAllContents(ActivityDiagramBehavior
        ))]
3   to [activityDB.name.toLower()/]
4   [for (aLocalVariable : LocalVariable | activityDB.
        eAllContents(LocalVariable)) separator('\n')]
```

```
5       let [aLocalVariable.generateArithimeticExpressions()/] [
            aLocalVariable.expression.generateFunctionCallExpressions
            ()/]
6       [/for]
7       [activityDB.start.generateControlNodes()/]
8       end
9       [/for]
10      [/template]
```

Template C.11 – The go procedure

```
1       [template public generateToGo(aModel:Model) ]
2       to go
3       [for (aPatch : Entity | aModel.eAllContents(InitSpace).
            entity)]
4       ask patches [ '[' /]
5       ask [aPatch.name.toLower()/]s[ '[' /]
6       [for (aNode : Node | aModel.mainBehavior.eAllContents(Node)
            ->asSequence())]
7       [if (aNode.oclIsTypeOf(DeclaredBehavior) and not aPatch.
            eAllContents(Behavior)->indexOf(aNode.oclAsType(
            DeclaredBehavior).behavior).oclIsUndefined() )]
8       [aNode.oclAsType(DeclaredBehavior).behavior.name.toLower()/]
            [']'/]
9       [/if]
10      [/for]
11      [ ']' /]
12      [/for]
13      [for (anInitEntity : Entity | aModel.eAllContents(InitEntity
            ).entity->asSet())]
14      ask [anInitEntity.name.toLower()/]s ['['/]
15      [for (aNode : Node | aModel.mainBehavior.eAllContents(Node)
            ->asSequence())]
16      [if (aNode.oclIsTypeOf(DeclaredBehavior) and not
            anInitEntity.eAllContents(Behavior)->indexOf(aNode.
            oclAsType(DeclaredBehavior).behavior).oclIsUndefined())]
17      [aNode.oclAsType(DeclaredBehavior).behavior.name.toLower()/]
18      [/if]
19      [/for]
20      [']'/]
21      [/for]
22      tick
23      end
24      [/template]
```

C.2 Cormas M2T templates

Template C.12 – Cormas classes generation

```
1  [comment Instances of all AgentLocation classes /]
2  [for (initSuperClass : Entity | m.eAllContents(InitEntity).
       entity->asSet())]
3  <class>
4  <Name>[initSuperClass.name.toUpperFirst()/]</Name>
5  <environment>CormasNS.Models.[m.name.toUpper()/]</
       environment>
6  <super>CormasNS.Kernel.AgentLocation</super>
7  <private>false</private>
8  <indexed-type>none</indexed-type>
9  <inst-vars>[for (att : AttributeClass | initSuperClass.
       attributes)][att.name.toLower()/] [/for]</inst-vars>
10 <class-inst-vars>[for (par : ParameterClass | initSuperClass
       .behavior.parameters)][par.name.toLower()/] [/for]</class
       -inst-vars>
11 <imports></imports>
12 <category>[m.name.toUpper()/]Category</category>
13 <attributes>
14 <package>[m.name.toUpper()/]</package>
15 </attributes>
16 </class>
17 [/for]
18 [comment Instances of SpatialEntityCell /]
19 [for (initSpace : InitSpace | m.eAllContents(InitSpace))]
20 <class>
21 <Name>[initSpace.entity.name.toUpperFirst()/]</Name>
22 <environment>CormasNS.Models.[m.name.toUpper()/]</
       environment>
23 <super>CormasNS.Kernel.SpatialEntityCell</super>
24 <private>false</private>
25 <indexed-type>none</indexed-type>
26 <inst-vars>[for (att : AttributeClass | initSpace.
       eAllContents(Assignment).variable->selectByKind(
       AttributeClass))][att.name.toLower()/] [/for]</inst-vars>
27 <class-inst-vars>[for (par : ParameterClass | initSpace.
       eAllContents(Assignment).variable->selectByKind(
       ParameterClass))][par.name.toLower()/] [/for]</class-inst
       -vars>
28 <imports></imports>
29 <category>[m.name.toUpper()/]Category</category>
30 <attributes>
31 <package>[m.name.toUpper()/]</package>
```

```
32      </attributes>
33      </class>
34      [/for]
35      <class>
36      <name>[m.name/]</name>
37      <environment>CormasNS.Models.[m.name.toUpper()/]</
            environment>
38      <super>CormasNS.Kernel.CormasModel</super>
39      <private>false</private>
40      <indexed-type>none</indexed-type>
41      <inst-vars>[for (initEntity : InitEntity | m.eAllContents(
            InitEntity))]the[initEntity.initName.toUpperFirst()/]s [
            initEntity.initName.toLower()/]InitialNumber [/for] [for
            (initSpace : InitSpace | m.eAllContents(InitSpace))]the[
            initSpace.entity.name.toUpperFirst()/]s[/for]</inst-vars>
42      <class-inst-vars></class-inst-vars>
43      <imports></imports>
44      <category>[m.name.toUpper()/]Category</category>
45      <attributes>
46      <package>[m.name.toUpper()/]</package>
47      </attributes>
```

Template C.13 – Code generation for accessing protocol methods in Cormas

```
1       [template public generateAccessingMainClass(m : Model)]
2       <methods>
3       <class-id>CormasNS.Models.[m.name.toUpper()/].[m.name.
            toUpper()/]</class-id> <category>accessing</category>
4       [for (create : InitEntity | m.eAllContents(InitEntity)->
            asSet())]
5       <body package="[m.name.toUpper()/]" selector="the[entities.
            name.toUpperFirst()/]">the[m.entities.name.toUpperFirst()
            /]s
6       ^self allTheEntities: [entities.name.toUpperFirst()/]
7       </body>
8
9       <body package="[m.name.toUpper()/]" selector="the[create.
            initName.toLowerFirst()/]s">the[create.initName/]s
10      ^the[create.initName/]s ifNil:['['/]the[create.initName/]s
            := IndexedSet new[']'/]
11      </body>
12
13      <body package="[m.name.toUpper()/]" selector='[create.
            initName/]InitialNumber">[create.initName/]InitialNumber
```

```
14      ^[create.initName/]InitialNumber : ifNil:['['/][create.
           initName/]InitialNumber := [create.initFunctionCall.
           oclAsType(FunctionCallExpression).
           generateInitialAgentNumbers()/][ ']' /]
15      </body>
16
17      <body package="[m.name.toUpper()/]" selector="[create.
           initName.toLowerFirst()/]InitialNumber">[create.initName.
           toLowerFirst()/]
18      InitialNumber: anObject[create.initName/]InitialNumber :=
           anObject
19      </body>
20      [/for]
21      </methods>
22      [/template]
```

Template C.14 – Methods code generation for the instance-creation protocol

```
1       [template public generateInstanceCreationMainClass(model :
           Model)]
2       <methods>
3       <class-id>CormasNS.Models.[model.name/].[model.name/]</class
           -id>
4       <category>instance-creation</category>
5       <body package="[model.name/]"selector="homogeneousEnv2">
6       homogeneousEnv2
7       self spaceModel initializeRegularX: [model.eAllContents(
           InitSpace).initFunctionCall.oclAsType(
           FunctionCallExpression).arguments->at(2).
           generateArithimeticExpressions()/] Y: [model.eAllContents
           (InitSpace).initFunctionCall.oclAsType(
           FunctionCallExpression).arguments->at(3).
           generateArithimeticExpressions()/] shape: #squared
           nbNeighbours: #eight boundaries: #toroidal.
8       [for (assign : Assignment | model.eAllContents(InitSpace).
           assingnments)]
9       [if (assign.variable.variableclass.oclIsTypeOf(
           AttributeClass))]
10      self the[model.eAllContents(InitSpace).entity.name/]s do:[
           '['/]: cell | cell init[assign.variable.variableclass.
           generateArithimeticExpressions().toUpperFirst()/][']'/].
11      [/if]
12      [/for]
13      self initAgents
14      </body>
15      <body package="[model.name/]" selector="initAgents">
16      initAgents
```

```
17      [for (initEntity : InitEntity | model.eAllContents(
            InitEntity))]
18      [if (initEntity.initName.oclIsUndefined())]
19      self createN: self [initEntity.entity.name.toLower()/]
            InitialNumber
20      randomlyLocatedAloneEntities[initEntity.entity.name.
            toUpperFirst()/][else]
21      self createN: self [initEntity.initName.toLower()/]
            InitialNumber randomlyLocatedAloneEntities:[initEntity.
            initName.toUpperFirst()/]
22      [/if]
23      [/for]
24      </body>
25      </methods>
26      [/template]
```

Template C.15 – Methods code generation for the init protocol

```
1       [template public generateInitMainClass(model : Model)]
2       <methods>
3       <class-id>CormasNS.Models.[model.name/].[model.name/]</class
            -id> <category>init</category>
4       <body package="[model.name/]" selector="noAgents">noAgents
5       self spaceModel loadEnvironmentFromFile: 'poor.env'
6       </body>
7       <body package="[model.name/]" selector="homogeneousEnv">
            homogeneousEnv
8       self spaceModel loadEnvironmentFromFile: 'poor.env'.
9       self initAgents
10      </body>
11      <body package="[model.name/]" selector="fragmentedEnv">
            fragmentedEnv
12      self spaceModel loadEnvironmentFromFile: 'fragmented.env'.
13      self initAgents
14      </body>
15      </methods>
16      [/template]
```

Template C.16 – Method code generation for the control protocol

```
1       [template public generateControlMainClass(model : Model)]
2       <methods>
3       <class-id>CormasNS.Models.[model.name/].[model.name/]</class
            -id> <category>control</category>
4       <body package="[model.name/]" selector="step">
5       step: t
```

```
6       [for (initSpace : InitSpace | model.eAllContents(InitSpace))
           ]
7       self stepEntities: self the[initSpace.initFunctionCall.
           oclAsType(FunctionCallExpression).arguments->first().
           generateArithimeticExpressions().toUpperFirst()/]s.
8       [/for]
9       [for (initEntity : InitEntity | model.eContents(InitEntity)
           ->asSet())]
10      self askRandom: [initEntity.initFunctionCall.oclAsType(
           FunctionCallExpression).arguments->first().
           generateArithimeticExpressions().toUpperFirst()/] toDo: #
           step
11      </body>
12      [/for]
13      </methods>
14      [/template]
```

Template C.17 – Methods code generation for the probes protocol

```
1       [template public generateProbesMainClass(model : Model)]
2       <methods>
3       <class-id>CormasNS.Models.[model.name.toUpper()/].[model.
           name.toUpper()/]</class-id> <category>probes</category>
4       [for (initEntity : InitEntity | model.eAllContents(
           InitEntity)->asSet())]
5       <body package="[model.name.toUpper()/]" selector="[
           initEntity.initName.toLower()/]Size">[initEntity.initName
           /]Size
6       ^self the[initEntity.initName.toUpperFirst()/]s size</body>
7       [/for]
8       [for (initSpace : InitSpace | model.eAllContents(InitSpace)
           ->asSet())]
9       [for (att : AttributeClass | initSpace.entity.attributes)]
10      <body package="[model.name.toUpper()/]" selector="[initSpace
           .entity.name.toLower()/][att.name.toUpperFirst()/]">[
           initSpace.entity.name.toLower()/][att.name.toUpperFirst()
           /]
11      ^self the[initSpace.entity.name.toUpperFirst()/]s size
12      </body>
13      [/for]
14      [/for]
15      </methods>
16      [/template]
```

Template C.18 – Model behaviors for custom protocols

```
1       [template public generateCustomProtocol(m : Model)]
```

```
2       <methods>
3       <class-id>CormasNS.Models.[m.name.toLower()/].[m.eContents(
            Entity).name.toLower()/]</class-id> <category>
            GeneratedProtocol</category>
4       [for (be : Behavior | m.eAllContents(Behavior))]
5       <body package="[m.name.toLower()/]" selector="[be.name.
            toLower()/]">[be.name.toLower()/]
6       </body>
7       [/for]
8       </methods>
9       [/template]
```

Bibliography

Andersen, David F. et al. (2007). "Group Model Building: Problem Structing, Policy Simulation and Decision Support". In: *The Journal of the Operational Research Society* 58.5, pp. 691–694.

Anderson, J (1996). "A simple theory of complex cognition". In: *American Psychologist* 51.4, pp. 355–365.

Anderson, J R (1983). *The Architecture of Cognition*. Harvard University Press.

Andova, Suzana, Mark G J Van Den Brand, and Luc Engelen (2012). "Reusable and correct endogenous model transformations". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7307 LNCS. Springer Berlin Heidelberg, pp. 72–88.

Aubert, Sigrid and Jean-Pierre Müller (2013). "Incorporating institutions, norms and territories in a generic model to simulate the management of renewable resources". In: *Artificial Intelligence and Law* 21.1, pp. 47–78.

Aubert, Sigrid, Jean-Pierre Müller, and Julliard Ralihalizara (2010). "MIRANA: a socio-ecological model for assessing sustainability of community-based regulations". In: *International Congress on Environmental Modelling and Software*, p. 9.

Bandini, Stefania, Sara Manzoni, and G Vizzari (2009). "Agent Based Modeling and Simulation : An Informatics Perspective". en. In: *Journal of Artificial Societies and Social Simulation* 12.4, p. 4.

Barendrecht, P.J. (2010). *Modeling transformations using QVT Operational Mappings*. Tech. rep. Eindhoven University of Technology, p. 47.

Barreteau, O., F. Bousquet, and J.-M. Attonaty (2000). *Role-playing games for opening the black box of multi-agent systems: method and lessons of its application to Senegal River*. en. (Last accessed: 05/16/2014).

Benjamin, D Paul, Pace Plaza, and New York (2001). "ADAPT : A Cognitive Architecture for Robotics An Implementation of ADAPT". In: *Forum American Bar Association*, pp. 337–338.

Beydeda, Sami, Matthias Book, and Volker Gruhn (2005). *Model-driven software development*. Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–464.

Blackwell, Alan F. (2001). "Introduction: Thinking with diagrams". en. In: *Artificial Intelligence Review* 15.1-2. Ed. by Alan F. Blackwell, pp. 1–3.

Bommel, Pierre and Francisco Dieguez (2011). "One more step towards participatory modeling: Involving local stakeholders in designing scientific models for participative foresight studies". In: *Proceedings of the 2011 European Social Simulation Association Conference*, pp. 19–23.

Bommel, Pierre, Francisco Dieguez, et al. (2014). "A further step towards participatory modelling. fostering stakeholder involvement in designing models by using executable UML". In: *JASSS* 17.1, p. 6.

Borshchev, Andrei (2007). "Multi-Method Simulation Modeling using AnyLogic This presentation . . . " In:

Bousquet, François et al. (1998). "Cormas : Common-Pool Resources and Multi-agent Systems". In: *Tasks and Methods in Applied Artificial Intelligence* 1416.December, pp. 826–837.

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool, pp. 182–.

Cabot, Jordi (2009). *Model-based Engineering vs Model-Driven Engineering*. URL: http://modeling-languages.com/model-based-engineering-vs-model-driven-engineering-2/ (Last accessed: 05/19/2016).

Cardwell, Hal, Stacy Langsdale, and Kurt Stephenson (2009). *The Shared Vision Planning Primer : How to incorporate*. English. Tech. rep. January, pp. 11–38.

Challenger, Moharram et al. (2014). "On the use of a domain-specific modeling language in the development of multiagent systems". In: *Engineering Applications of Artificial Intelligence* 28, pp. 111–141.

Chu, Thanh Quang et al. (2012). "Towards a methodology for the participatory design of agent-based models". In: *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7057 LNAI, pp. 428–442.

D'Aquino, Patrick et al. (2002). "The Role Playing Games in an ABM participatory modeling process: outcomes from five different experiments carried out in the last five years". In: *Integrated Assessment and Decision Support,1st Biennial Meeting of the International Environmental Modelling and Software Society*, pp. 275–280.

Demazeau, Yves and Jean-Pierre Müller (1990). "Decentralized A.I. : proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Cambridge, England, August 16-18, 1989". In: pp. viii, 263.

Demirkol, Sebla et al. (2013). "A DSL for the development of software agents working within a semantic web environment". In: *Computer Science and Information Systems* 10.4 SPEC.ISSUE, pp. 1525–1556.

Dent, Valeda F. (2007). "Intelligent agent concepts in the modern library". In: *Library Hi Tech* 25.1. Ed. by Kenneth Einar Himma, pp. 108–125.

Diaw, Samba, Redouane Lbath, and Bernard Coulette (2010). "Etat de l'art sur le développement logiciel basé sur les transformations de modèles". In: *Numéro spécial TSI - Ingénierie Dirigée par les Modèles* 29:4-5.4-5, p. 2.

Drogoul, Alexis (2015). "Agent-based modeling for multidisciplinary and participatory approaches to climate change adaptation planning". In: *RFCC (Regional Forum on Climate Change)*.

Eclipse Foundation (2014). *Eclipse Modeling Project*. URL: http://www.eclipse.org/modeling/ (Last accessed: 07/04/2016).

– (2016[a]). *Ecore API*. URL: http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html (Last accessed: 07/04/2016).

– (2016[b]). *Graphical Modeling Framework Tutorial - Part 1*. URL: https://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1 (Last accessed: 07/11/2016).

Elden, S. (2013). "The Significance of Territory". In: *Geographica Helvetica* 68, pp. 65–68.

Epstein, Joshua M. (2002). "Modeling civil violence: an agent-based computational approach." In: *Proceedings of the National Academy of Sciences of the United States of America* 99.3, pp. 7243–7250.

Etienne, Michel (2014). *Companion Modelling - A participatory Approach to Support Sustainable Development*. en. Editions Quae, pp. 1–403.

Etienne, Michel, Derick R. du Toit, and Sharon Pollard (2011). "ARDI: A co-construction method for participatory modeling in natural resources management". In: *Ecology and Society* 16.1.

European PhD School on Robotic Systems (2016). *Model-Driven Engineering and Knowledge representation*. URL: http://www.phdschoolinrobotics.eu/ContentMDE.html (Last accessed: 06/17/2016).

Ferber, Jacques (1999). *Multi-agent Systems: An Introduction to Distributed Artificial Intelligence*, p. 509.

Flater, David, Philippe Martin, and Michelle Crane (2009). "Rendering UML Activity Diagrams as Human-Readable Text." In: *Ike*, pp. 207–213.

Fowler, Martin (2010). *Domain-Specific Languages*. Vol. 5658. Addison-Wesley Professional, p. 640.

Franklin, Stan and Art Graesser (1997). "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". In: *Intelligent agents III agent theories, architectures, and languages*, pp. 21–35.

Frigg, Roman and Stephan Hartmann (2012). "Models in Science". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Fall 2012.

Galvão, Ismênia and Arda Goknil (2007). "Survey of traceability approaches in model-driven engineering". In: *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pp. 313–324.

Gaube, Veronika et al. (2006). "Linking agent-based models with stock and flow models: Impacts of subsidy policy an farmer housholds, land use and nutrient flow at regional level". In: *ConAccount Conference*. Vol. 35. 2. Vienna, p. 2.

Ghosh, Debasish (2010). *DSLs in Action*. Manning, pp. 1–377.

Gourmelon, Françoise et al. (2013). "Role-playing game developed from a modelling process: A relevant participatory tool for sustainable development?

A co-construction experiment in an insular biosphere reserve". In: *Land Use Policy* 32, pp. 96–107.

Gronback, Richard C (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1st ed. Addison-Wesley Professional.

Hostler, R. Eric, Victoria Y. Yoon, and Tor Guimaraes (2005). "Assessing the impact of internet agent on end users' performance". In: *Decision Support Systems* 41.1, pp. 313–323.

Information technology Syntactic metalanguage, Extended BNF (1996). *ISO/IEC 14977*. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip (Last accessed: 05/19/2016).

Jäger, Gerhard and James Rogers (2012). "Formal language theory: refining the Chomsky hierarchy." In: *Philosophical transactions of the Royal Society of London. Series B, Biological sciences* 367.1598, pp. 1956–70.

Jan Köhnlein (2009). *Domain-Specific Langauges*. URL: http://www.slideshare.net/meysholdt/converging-textual-and-graphical-editors (Last accessed: 07/06/2016).

Jarrah, Moath et al. (2015). "A Multi-Agent Simulation Framework to Support Agent Interactions under Different Domains". In: pp. 211–223.

Jouault, Frédéric, Jean Bézivin, and Ivan Kurtev (2006). "TCS: a DSL for the specification of textual concrete syntaxes in model engineering". In: *Proceedings of the 5th international conference on Generative programming and component engineering - GPCE '06*, pp. 1–6.

Kappel, Gerti et al. (2012). "Model transformation by-example: A survey of the first wave". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7260 LNCS, pp. 197–215.

Kent, Stuart (2002). "Model Driven Engineering". In: *Integrated Formal Methods* 2335.2, pp. 286–298.

Kieras, David E., Scott D. Wood, and David E. Meyer (1997). "Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task". In: *ACM Transactions on Computer-Human Interaction* 4.3, pp. 230–275.

Kleppe, Anneke, J Warmer, and Wim. Bast (2003). "The Model Driven Architecture: Practice and Promise. 2003". In: *Addison Wesley*, p. 170.

Kosar, Tomaž et al. (2010). "Comparing general-purpose and domain-specific languages: An empirical study". In: *Computer Science and Information Systems* 7.2, pp. 247–264.

Le Moigne, Jean-Louis (1990). *La modélisation des systemes complexes*.

Mabrouki, Olfa (2015). "Semantic Framework for Managing Privacy Policies in Ambient Intelligence Olfa Mabrouki Semantic Framework For Managing Privacy Policies In Ambient Intelligence". In:

Maharaj, Savi, Tamsin Mccaldin, and Adam Kleczkowski (2011). "A Participatory Simulation Model for Studying Attitudes to Infection Risk". In: *SCSC '11: Proceedings of the 2011 Summer Computer Simulation Conference*, pp. 8–13.

Mayer, I. S. (2009). "The Gaming of Policy and the Politics of Gaming: A Review". In: *Simulation & Gaming* 40.6, pp. 825–862.

McGinnis, Michael D. and Elinor Ostrom (2014). "Social-ecological system framework: Initial changes and continuing challenges". In: *Ecology and Society* 19.2, art30.

Meadows, Dennis, John Sterman, and Andrew King (2015). *Fishbanks: A Renewable Resource Management Simulation*. URL: https://mitsloan.mit.edu/LearningEdge/simulations/fishbanks/Pages/fish-banks.aspx.

Mens, Tom and Pieter Van Gorp (2006). "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152.1-2, pp. 125–142.

Mernik, Marjan, Jan Heering, and Anthony M. Sloane (2005). "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4, pp. 316–344.

Michel, Fabien et al. (2011). "Situational programming: Agent behavior visual programming for MABS novices". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6532 LNAI. Springer, pp. 1–15.

Michell, T M (1997). "Machine Learning". In:

Morand, Bernard (2000). "Le processus de représentation, un cadre préliminaire pour une approche expérimentale du cas des diagrammes." In: *Journées francophones d'ingénierie des connaissances*. Toulouse,France, pp. 73–81.

Newell, Allen (1992). *Précis of Unified theories of cognition*. Vol. 15. Harvard University Press, pp. 425–492.

Nilsson, Nils J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, p. 513.

OMG (2014). *Model Driven Architecrure - Guide revision 2.0*. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

– (2015). *Meta Object Facility(MOF) Core Specification*. URL: http://www.omg.org/spec/MOF/2.5/.

Page, Christophe le et al. (2012). *Participatory agent-based simulation for renewable resource management: The role of the cormas simulation platform to nurture a community of practice*.

Pastor, Oscar et al. (2008). "Model-driven development". In: *Informatik-Spektrum* 31.5, pp. 394–407.

Pattis, Richard (2013). "Chapter 1 EBNF : A Notation to Describe Syntax". In: pp. 1–19.

Pepper, J. W. and B. B. Smuts (2000). "The evolution of cooperation in an ecological context: an agent-based mode". In: *Dynamics of human and primate societies: agent-based modeling of social and spatial processes.* Oxford University Press,Oxford, pp. 45–76.

Prieto-Diaz, Ruben (1990). "Domain Analysis: An Introduction". In: *Software Engineering Notes* 15.2, pp. 47–54.

Promburom, P (2002). "Participatory Multi-agent Systems Modeling for Collective Watershed Management: The Use of Role Playing Game." In: *Management*.

Ramsey, Kevin (2009). "GIS, modeling, and politics: On the tensions of collaborative decision support". In: *Journal of Environmental Management* 90.6, pp. 1972–1980.

Renger, Michiel, Gwendolyn L. Kolfschoten, and Gert Jan De Vreede (2008). "Challenges in collaborative modelling: a literature review and research agenda".

In: *International Journal of Simulation and Process Modelling*. Ed. by Jan L. G. Dietz, Antonia Albani, and Joseph Barjis. Vol. 4. Lecture Notes in Business Information Processing 3/4. Springer Berlin Heidelberg, p. 248.

Rodrigues Da Silva, Alberto (2015). "Model-driven engineering: A survey supported by the unified conceptual model". In: *Computer Languages, Systems and Structures* 43, pp. 139–155.

Ron, Sun (2006). "The CLARION cognitive architecture: Extending cognitive modeling to social simulation". In: *Cognition and MultiAgent Interaction*, pp. 79–99.

Rothenberg, Jeff et al. (1989). "The Nature of Modeling". In: *in Artificial Intelligence, Simulation and Modeling*. John Wiley & Sons, pp. 75–92.

Russell, Stuart, Peter Norvig, and Artificial Intelligence (1995). "A modern approach". In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25, p. 498.

Sacevski, Igor and Jadranka Veseli (2007). "Introduction to Model Driven Architecture (MDA)". In: June, pp. 1–15.

Schmidt, Douglas C. (2006). "Model-driven engineering". In: *Computer* 39.2, pp. 25–31.

Schürr, Andy (1994). "Specification of Graph Translators with Triple Graph Grammars". In: *WG 1994*. Vol. 903. Springer Berlin Heidelberg, pp. 151–163.

Selic, Bran (2003). "The pragmatics of model-driven development". In: *IEEE Software* 20.5, pp. 19–25.

– (2004). "On the Semantic Foundations of Standard UML 2.0". In: *Formal Methods for the Design of Real-Time Systems*, pp. 181–199.

Sowa, John F (2011). "Cognitive Architectures For Conceptual Structures". In: *Proceedings of the 19th international conference on Conceptual structures for discovering knowledge*, pp. 35–49.

Stahl, Thomas, Markus Voelter, and Krzysztof Czarnecki (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Steinberg, David et al. (2009). *EMF: Eclipse Modeling Framework 2.0*, p. 704.

Tàbara, J David et al. (2007). "Participatory Modelling For The Integrated Sustainability Assessment Of Water: the World Cellular Model and the MATISSE Project". In: *Integrated Assessment*, pp. 1–29.

Taillandier, Patrick (2014). "Traffic simulation with the GAMA platform". In: *International Workshop on Agents in Traffic and Transportation*.

Taillandier, Patrick et al. (2012). "GAMA: A simulation platform that integrates geographical information data, agent-based modeling and multi-scale control". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7057 LNAI. Springer, pp. 242–258.

Touraille, Luc, David R.C. Hill, and Mamadou K. Traore (2012). "Application of Model-Driven Engineering and Metaprogramming to DEVS Modeling & Simulation". English. PhD thesis. Clermont-Ferrand, p. 312.

Van Deursen, Arie, Paul Klint, and Joost Visser (2000). "Domain-specific languages". In: *Centrum voor Wiskunde en Informatika* 35.6, pp. 26–36.

Voinov, Alexey and Francois Bousquet (2010). "Modelling with stakeholders". In: *Environmental Modelling and Software* 25.11, pp. 1268–1281.

Wilensky, Uri (1999). *NetLogo: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL*. en. URL: http://jmvidal.cse.sc.edu/lib/netlogo.html (Last accessed: 05/13/2016).

– (2004). *NetLogo Rebellion model*. URL: http://ccl.northwestern.edu/netlogo/models/Rebellion%20http://www.cs.sjsu.edu/~pearce/modules/lectures/nlogo/library/Rebellion.htm (Last accessed: 08/08/2016).

Williams, G C (1966). "Adaptation and Natural Selection: A Critique of Some Current Evolutionary Thought. Princeton: Princeton University Press". In: p. 307.

Wooldridge, Michael (2009). *An Introduction to MultiAgent Systems*. 2nd. Wiley Publishing.

Wooldridge, Michael, Jörg P Müller, and Milind Tambe (1995). "Agent theories, architectures, and languages: A bibliography". In: *Intelligent Agents II Agent Theories, Architectures, and Languages* 890, pp. 408–431.

Zeigler, Bernard P. and Hessam S Sarjoughian (2003). "Introduction to devs modeling and simulation with java: Developing component-based simulation models". In: *Technical Document, University of Arizona*.